

单片机C语言编程与 Proteus仿真技术

徐爱钧 著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书在介绍8051单片机组成原理的基础上,结合目前流行的Keil C51编译器和Proteus虚拟仿真环境,阐述单片机C语言编程和虚拟仿真应用技术,包括单片机中断系统、定时器/计数器、串行口等片内资源的工作原理、单片机系统扩展、DAC与ADC、键盘与显示器接口技术,详细介绍了单片机片内、片外资源的C语言程序设计及其Proteus虚拟仿真应用方法,给出了大量单片机C语言程序范例和Proteus原理电路图,所有范例均在Proteus软件平台上调试通过,可以直接运行。

为帮助读者更好地学习和掌握C51应用编程方法和Proteus仿真技术,本书附赠一张CD-ROM光盘,包含Keil公司全功能C51评估软件包、Proteus仿真电路图及各章所有范例的程序源代码。

本书适合从事单片机应用系统开发研制的广大工程技术人员阅读,也可以作为高等院校相关专业大学生或研究生的教学参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

单片机C语言编程与Proteus仿真技术 / 徐爱钧著. —北京:电子工业出版社, 2016.1

ISBN 978-7-121-27538-8

I. ①单… II. ①徐… III. ①单片微型计算机—C语言—程序设计 ②单片微型计算机—系统仿真—应用软件 IV. ①TP312 ②P368.1

中国版本图书馆CIP数据核字(2015)第267039号

责任编辑:富 军

印 刷:三河市华成印务有限公司

装 订:三河市华成印务有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编 100036

开 本:787×1092 1/16 印张:22.5 字数:576千字

版 次:2016年1月第1版

印 次:2016年1月第1次印刷

印 数:3 000册 定价:59.00元(含光盘1张)

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。



P R E F A C E

8051是目前国内外工业测量控制领域内使用极为广泛的一类8位单片机。它在一块芯片上同时集成CPU、ROM、RAM及多种外围功能接口,具有体积小、价格低、功能强、可靠性高、使用方便灵活等特点,以单片机为核心设计各种智能化电子设备,周期短,成本低,易于更新换代,维修方便,已成为电子设计中最为普遍的应用手段。世界上许多大半导体厂商,如Atmel、Analog Device、Infineon、NXP、TI、SiLAB等公司都推出了各具特色的8051系列单片机。

早期,单片机应用开发大多采用汇编语言编程,编程效率不高,程序不易移植和维护。随着Keil C51编译器的流行,现在已经普遍采用C语言进行单片机应用编程。C语言具有类似自然语言的特点,既能直接操作机器硬件,又可以进行方便灵活的高级语言编程。在单片机应用系统开发过程中,除了编程工具之外,硬件平台也必不可少。目前,各种单片机开发平台层出不穷,英国Labcenter公司推出的Proteus软件是一款极好的单片机虚拟硬件平台,以其特有的仿真技术很好地解决了单片机及其外围电路的设计和协同仿真问题,可以在没有单片机实际硬件的条件下,利用PC机进行虚拟仿真实现单片机系统的软、硬件设计。Proteus虚拟硬件平台可以与Keil C51完美结合,在原理图中直接进行单片机C语言程序的源代码仿真调试,实现对系统性能的综合评估,验证各项技术指标。Proteus平台涵盖了8051等多种微处理器模型及各种常用电子元器件,包括74系列、CMOS4000系列集成电路、A/D和D/A转换器、键盘、LCD显示器、LED显示器,还提供示波器、逻辑分析仪、通信终端、电压/电流表、I²C/SPI终端等各种虚拟仪表,可以直接用于虚拟仿真,结合原理图和源码级程序调试,能够立即观察到单片机应用系统的输入、输出效果,极大地提高了应用系统的设计效率。

本书在构思及选材上符合单片机应用发展要求,突出先进性和实用性,对C51应用编程方法、Proteus虚拟仿真技术等进行详尽阐述,给出了大量单片机C语言程序和Proteus仿真设计范例。所有范例均已在Proteus平台上调试通过,可以直接运行。

全书共9章:

第1章阐述8051单片机基本组成、存储器结构、CPU时序、并行I/O端口及指令系统。

第2章阐述Proteus虚拟硬件平台,介绍在ISIS集成环境中绘制原理电路图、与Keil C51联机实现源代码仿真调试的方法。

第3章阐述Keil C51应用程序设计,介绍C51的基本语句、数据类型、Keil C51对ANSI C的扩展、与汇编语言程序接口及C51库函数等。

第4章阐述单片机片内资源应用,介绍C51编程的基本原则方法,给出中断系统、定时器/计数器、串行口等功能部件的C语言应用编程实例。

第5章阐述系统扩展与低功耗应用,介绍存储器、并行I/O端口的扩展及单片机低功耗应用方法,给出并行接口扩展芯片和低功耗工作方式的C语言应用编程实例。

第6章阐述键盘与显示器接口应用,介绍矩阵接盘、数码管、点阵字符和图型液晶显示

器等与单片机的接口方法，给出C语言应用编程实例。

第7章阐述数模与模数转换接口应用，介绍传统并行及新型串行D/A、A/D转换器芯片及其与单片机的接口方法，给出C语言应用编程实例。

第8章阐述I²C总线接口应用，介绍I²C总线结构与数据传输，给出I²C接口存储器芯片、A/D-D/A转换芯片及时钟芯片的C语言应用编程实例。

第9章给出5个单片机Proteus虚拟仿真设计实例及其完整的C51源程序。

本书在编写过程中得到广州风标信息技术有限公司（<http://www.windway.cn>）匡载华总经理的大力支持和热情帮助，电子工业出版社柴燕、富军编辑提出了许多宝贵意见，徐阳、彭秀华等参加了部分章节的编写和程序调试工作，在此一并表示感谢。

由于作者水平有限，书中难免会有错误和不妥之处，恳请广大读者批评指正，读者可通过电子邮件：ajxu@tom.com、ajxu41@sohu.com直接与作者联系。Proteus的DEMO软件可到官方网站<http://www.labcenter.co.uk>下载，或者与国内代理商广州风标信息技术有限公司联系购买正版软件。

徐爱钧 于长江大学



CONTENTS

第1章 8051单片机基础	1
1.1 8051单片机的特点与基本结构	1
1.2 8051单片机的存储器结构	5
1.3 CPU时序	8
1.4 复位信号与复位电路	10
1.5 并行I/O端口结构	11
1.6 指令系统	13
1.7 指令的寻址方式	15
1.7.1 寄存器寻址	15
1.7.2 直接寻址	15
1.7.3 立即寻址	15
1.7.4 寄存器间接寻址	16
1.7.5 变址寻址	16
1.7.6 相对寻址	17
1.7.7 位寻址	18
1.8 指令分类详解	18
1.8.1 算术运算指令	18
1.8.2 逻辑运算指令	20
1.8.3 数据传送指令	21
1.8.4 控制转移指令	23
1.8.5 位操作指令	25
1.9 汇编语言程序设计	26
第2章 Proteus虚拟仿真	31
2.1 集成环境ISIS	31
2.2 绘制原理图	35
2.3 创建汇编语言源代码仿真文件	37
2.4 在原理图中进行源代码仿真调试	39
2.5 原理图与Keil环境联机仿真调试	42
第3章 Keil C51应用程序设计	49
3.1 Keil C51程序设计的基本语法	49
3.1.1 Keil C51程序的一般结构	49
3.1.2 数据类型	50

3.1.3	常量、变量及其存储模式	51
3.1.4	运算符与表达式	52
3.2	C51程序的基本语句	56
3.2.1	表达式语句	56
3.2.2	复合语句	56
3.2.3	条件语句	56
3.2.4	开关语句	57
3.2.5	循环语句	57
3.2.6	goto、break、continue语句	58
3.2.7	返回语句	58
3.3	函数	59
3.3.1	函数的定义与调用	59
3.3.2	中断服务函数与寄存器组定义	60
3.4	Keil C51编译器对ANSI C的扩展	61
3.4.1	存储器类型与编译模式	61
3.4.2	关于bit、sbit、sfr、sfr16数据类型	62
3.4.3	一般指针与基于存储器的指针及其转换	65
3.4.4	C51编译器对ANSI C函数定义的扩展	66
3.5	C51编译器的数据调用协议	69
3.5.1	数据在内存中的存储格式	69
3.5.2	目标代码的段管理	71
3.6	与汇编语言程序的接口	73
3.6.1	参数传递规则	73
3.6.2	C51与汇编语言混合编程举例	77
3.7	绝对地址访问	80
3.7.1	采用扩展关键字“_at_”或指针定义变量的绝对地址	80
3.7.2	采用预定义宏指定变量的绝对地址	81
3.8	Keil C51库函数	81
3.8.1	本征库函数	82
3.8.2	字符判断转换库函数	82
3.8.3	输入、输出库函数	83
3.8.4	字符串处理库函数	87
3.8.5	类型转换及内存分配库函数	88
3.8.6	数学计算库函数	89
第4章	单片机片内资源应用	91
4.1	采用Keil C51编写应用程序的基本原则	91
4.2	并行I/O端口	92
4.2.1	典型单片机输入、输出电路	92
4.2.2	单片机I/O端口应用编程	94

4.3	中断系统	100
4.3.1	中断系统结构与中断控制	101
4.3.2	中断响应	104
4.3.3	中断系统应用编程	106
4.4	定时器/计数器	110
4.4.1	定时器/计数器的工作方式与控制	110
4.4.2	定时器方式应用编程	113
4.4.3	计数器方式应用编程	119
4.4.4	利用定时器产生音乐	121
4.5	串行口	124
4.5.1	串行口的工作方式与控制	125
4.5.2	串口/并口转换应用编程	128
4.5.3	单片机与PC机通信应用编程	130
4.5.4	单片机与单片机通信应用编程	132
4.5.5	修改底层函数实现printf()重新定向	139
第5章	系统扩展与低功耗应用	143
5.1	存储器扩展	143
5.1.1	程序存储器扩展	143
5.1.2	数据存储器扩展	144
5.2	并行I/O端口扩展	146
5.2.1	线选法	146
5.2.2	地址译码法	147
5.2.3	8155和8255并行接口扩展芯片应用编程	149
5.3	8051单片机的低功耗应用	158
5.3.1	空闲工作方式	158
5.3.2	掉电工作方式	159
5.3.3	低功耗方式应用编程	159
第6章	键盘与显示器接口应用	163
6.1	LED显示器接口技术	163
6.1.1	七段LED数码管显示器	163
6.1.2	单个74HC595驱动多位LED数码管	168
6.1.3	串行接口8位共阴极LED驱动器MAX7219	171
6.2	键盘接口技术	177
6.2.1	编码键盘接口	178
6.2.2	非编码键盘接口	180
6.3	8279可编程键盘/显示器芯片接口技术	183
6.3.1	8279的引脚排列	183
6.3.2	8279的数据输入、显示输出及命令格式	184

6.3.3	8279接口应用编程	189
6.4	点阵字符型LCD接口技术	191
6.4.1	点阵字符型LCD显示模块	192
6.4.2	直接方式接口应用编程	197
6.4.3	间接方式接口应用编程	200
6.4.4	4位数据总线接口应用编程	203
6.5	12864点阵图型LCD接口技术	206
6.5.1	12864点阵图型LCD显示模块	206
6.5.2	12864 LCD接口应用编程	209
6.6	T6963点阵图型LCD接口技术	213
6.6.1	T6963点阵图型LCD显示模块	213
6.6.2	T6963 LCD接口应用编程	216
第7章	模数与数模转换接口应用	223
7.1	转换器的主要技术指标	223
7.2	数/模转换器DAC接口技术	224
7.2.1	DAC0832接口应用编程	225
7.2.2	DAC1208接口应用编程	229
7.2.3	串行D/A芯片TLC5615接口应用编程	231
7.2.4	利用DAC接口实现波形发生器	233
7.3	模/数转换器ADC接口技术	238
7.3.1	比较式ADC0809接口应用编程	239
7.3.2	积分式ADC ICL7135接口应用编程	243
7.3.3	串行A/D芯片TLC549接口应用编程	248
第8章	I ² C总线接口应用	253
8.1	I ² C总线简介	253
8.2	I ² C总线结构与数据传输	253
8.3	I ² C总线通用驱动程序	257
8.4	I ² C接口存储器芯片24C04应用编程	258
8.5	I ² C接口A/D-D/A转换芯片PCF8591应用编程	263
8.6	I ² C接口时钟芯片PCF8563应用编程	274
第9章	Proteus仿真设计实例	285
9.1	红外遥控系统	285
9.1.1	功能要求	285
9.1.2	硬件电路设计	285
9.1.3	软件程序设计	285
9.2	点阵LED显示屏	296
9.2.1	功能要求	296

9.2.2 硬件电路设计	296
9.2.3 软件程序设计	297
9.3 电子密码锁	299
9.3.1 功能要求	299
9.3.2 硬件电路设计	299
9.3.3 软件程序设计	300
9.4 DS18B20多点温度监测系统	316
9.4.1 功能要求	316
9.4.2 硬件电路设计	316
9.4.3 软件程序设计	320
9.5 SD卡WAV音频播放器	328
9.5.1 功能要求	328
9.5.2 硬件电路设计	329
9.5.3 软件程序设计	330
附录A 8051指令表	341
附录B Proteus中的常用元器件	347
参考文献	349

8051单片机基础

1.1 8051单片机的特点与基本结构

8051系列单片机是在美国Intel公司于20世纪80年代推出的MCS—51系列高性能8位单片机的基础上发展而来的。它在单一芯片内集成了并行I/O口、异步串行口、16位定时器/计数器、中断系统、片内RAM和片内ROM及其他一些功能部件。现在8051系列单片机已经有了很大的发展，除了Intel公司之外，Philips、Siemens、AMD、Fujitsu、OKI、Atmel、SST、Winbond等公司都推出了以8051为核心的新一代8位单片机。这种新型单片机的集成度更高，在片内集成了更多的功能部件，如A/D、PWM、PCA、WDT及高速I/O口等。不同公司推出的8051具有各自的功能特点，但它们的内核都是以Intel公司的MCS—51为基础的，并且指令系统兼容，从而给用户带来了广阔的选择范围，同时又可以采用相同的开发工具。

8051系列单片机在存储器的配置上采用所谓“哈佛”结构，即在物理上具有独立的程序存储器和数据存储器，而在逻辑上则具有相同的地址空间，利用不同的指令和寻址方式可分别访问64kB程序存储器空间和64kB数据存储器，共有111条指令。其中包括乘除指令和位操作指令。有5个中断源，分为2个优先级。8051单片机在片内RAM中开辟了4个工作寄存器区，每个区有8个通用寄存器，可以适用于多种中断或子程序嵌套的情况。在片内RAM中还开辟了1个位寻址区，利用位操作指令可以对位寻址区中每个单元的每一个位进行操作，特别适合于解决各种开关控制和逻辑问题。8051单片机在单芯片应用方式下，其4个并行I/O口P0~P3都可以作为输入、输出，在扩展应用方式下可采用P0和P2口作为片外扩展地址总线。8051单片机内部集成了两个16位定时器/计数器，可进行定时或计数操作，还集成了1个全双工的异步串行接口，为单片机之间相互通信或与上位机通信带来极大的方便。

8051单片机的基本组成如图1.1所示。一个单片机芯片内包括中央处理器CPU。它是单片机的核心，用于产生各种控制信号，并完成对数据的算术逻辑运算和传送。片内数据存储器RAM用于存放可以读/写的数据。片内程序存储器ROM用于存放程序指令或某些常数表格。四个8位的并行I/O接口P0、P1、P2和P3，每个口都可以用作输入或者输出。两个定时器/计数器T0和T1用于外部事件计数器，也可用于定时。内部中断系统具有5个中断源，两个优先级的嵌套中断结构，可实现二级中断服务程序嵌套，每一个中断源都可编程设定为高优先级

或低优先级中断。一个串行接口电路可用于异步接收发送器。内部时钟电路，其晶体和微调电容需要外接，振荡频率可以高达40MHz。以上各部分通过内部总线相连接。

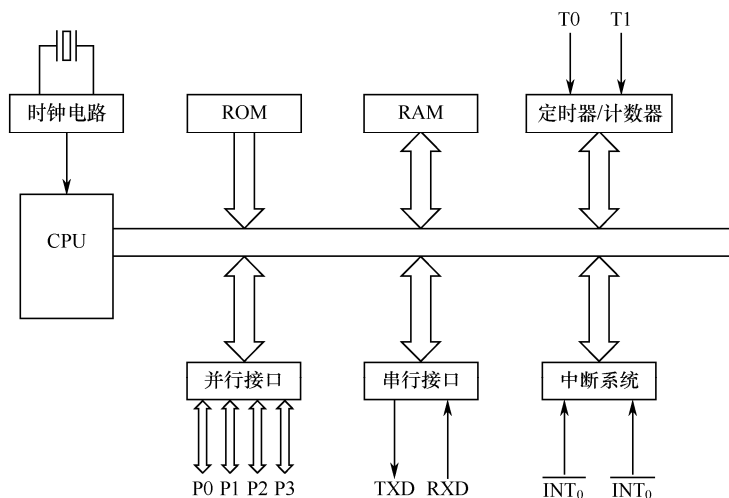


图1.1 8051单片机的基本组成

8051单片机没有单独的片外地址和数据总线，而是通过并行I/O端口的分时复用形成。P0口分时复用为低8位地址线和8位数据线。P2口则作为高8位地址线，从而形成16位地址线和8位数据线。一定要建立一个明确的概念，单片机在进行外部扩展时的地址线和数据线都不是独立的，而是与并行I/O端口公用的。这是8051单片机结构上的一个特点。

图1.2为8051单片机内部结构框图。其中，中央处理器CPU包含运算器和控制器两大部分；运算器完成各种算术和逻辑运算；控制器在单片机内部协调各功能部件之间的数据传送和运算操作，并对单片机外部发出若干控制信息。

1. 运算器

运算器以算术逻辑单元ALU为核心，加上累加器ACC、暂存寄存器TMP和程序状态字寄存器PSW等组成。ALU主要用于完成二进制数据的算术和逻辑运算，并通过对运算结果的判断影响程序状态字寄存器PSW中有关位的状态。累加器ACC是一个8位的寄存器（在指令中一般写为A），通过暂存寄存器TMP与ALU相连。ACC的工作最为繁忙，因为在进行算术逻辑运算时，ALU的一个输入多为ACC的输出，而大多数运算结果也需要送到ACC中，在做乘除运算时，B寄存器用来存放一个操作数，也用来存放乘除运算后的一部分结果，若不做乘除操作时，B寄存器可用作通用寄存器。程序状态字寄存器PSW也是一个8位寄存器，用于存放运算结果的一些特征，格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
CY	AC	F0	RS1	RS0	OV	\	P

其中各位的意义如下：

CY：进位标志。在进行加法或减法运算时，若运算结果的最高位有进位或借位，则CY=1，否则CY=0，在执行位操作指令时，CY作为位累加器。

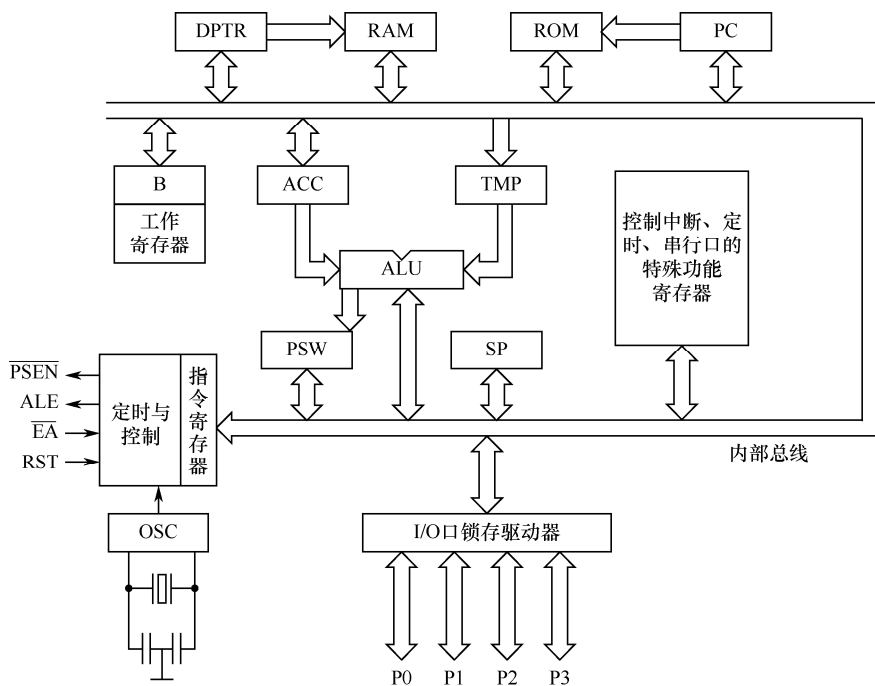


图1.2 8051单片机内部结构框图

AC：辅助进位标志。在进行加法或减法运算时，若低半字节向高半字节有进位或借位，则AC=1，否则AC=0，AC还作为BCD码运算调整时的判别位。

F0：用户标志。用户可根据自己的需要对F0赋以一定的含义，如可以用软件来测试F0的状态以控制程序的流向等。

RS1和RS0：工作寄存器组选择，可以用软件来置位或复位。它们与工作寄存器组的关系见表1.1。

表1.1 RS1和RS0与工作寄存器组的关系

RS1	RS0	工作寄存器组	片内 RAM 地址
0	0	第 0 组	00H~07H
0	1	第 1 组	08H~0FH
1	0	第 2 组	10H~17H
1	1	第 3 组	18H~1FH

OV：溢出标志。当两个带符号的单字节数进行运算，结果超出-128~+127的范围时，OV=1，表示有溢出，否则OV=0，表示无溢出。

PSW中的D1位为保留位。

P：奇偶校验标志。每条指令指行完毕后，都按照累加器A中“1”的个数来决定P值，当“1”的个数为奇数时，P=1，否则P=0。

2. 控制器

控制器包括定时控制逻辑、指令寄存器、指令译码器、程序计数器PC、数据指针寄存

器DPTR、堆栈指针SP、地址寄存器和地址缓冲器等。它的功能是对逐条指令进行译码，并通过定时和控制电路在规定的时刻发出各种操作所需的内部和外部控制信号协调各部分的工作。下面简单介绍其中主要部件的功能。

程序计数器PC：用于存放下一条将要执行指令的地址。当一条指令按PC所指向的地址从程序存储器中取出之后，PC的值会自动增量，即指向下一条指令。

堆栈指针SP：用来指示堆栈的起始地址。8051单片机的堆栈位于片内RAM中，而且属于“上长型”堆栈，复位后，SP被初始化为07H，使得堆栈实际上由08H单元开始。必要时可以给SP装入其他值，重新规定栈底的位置。堆栈中数据操作规则是“先进后出”，每往堆栈中压入一个数据，SP的内容自动加1，随着数据的压入，SP的值将越来越大，当数据从堆栈弹出时，SP的值将越来越小。

指令译码器：当指令送入指令译码器后，由译码器对该指令进行译码，即把指令转变成所需要的电平信号，CPU根据译码器输出的电平信号使定时控制电路产生执行该指令所需要的各种控制信号。

数据指针寄存器DPTR：是一个16位寄存器，由高位字节DPH和低位字节DPL组成，用来存放16位数据存储器的地址，以便对片外64kB的数据RAM区进行读写操作。

采用40引脚双列直插DIP封装的8051单片机引脚分配图如图1.3所示。

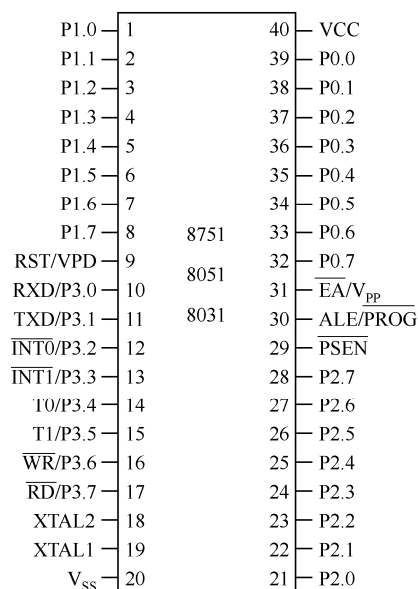


图1.3 8051单片机引脚分配图

各引脚功能如下：

RST/VPD (9)：RST是复位信号输入端。当此输入端保持两个机器周期（24个振荡周期）的高电平时，就可以完成复位操作。第二功能是VPD，即备用电源输入端，当主电源发生故障，降低到规定的低电平以下时，VPD将为片内RAM提供备用电源，以保证存储在RAM中的信息不丢失。

XTAL2 (18) 和XTAL1 (19)：在使用单片机内部振荡电路时，这两个端子用来外接石英晶体和微调电容（图1.4 (a)）。在使用外部时钟时，用来输入时钟脉冲，但对NMOS和

CMOS芯片接法不同,图1.4 (b) 为NMOS芯片8051外接时钟,图1.4 (c) 为CMOS芯片8051外接时钟。

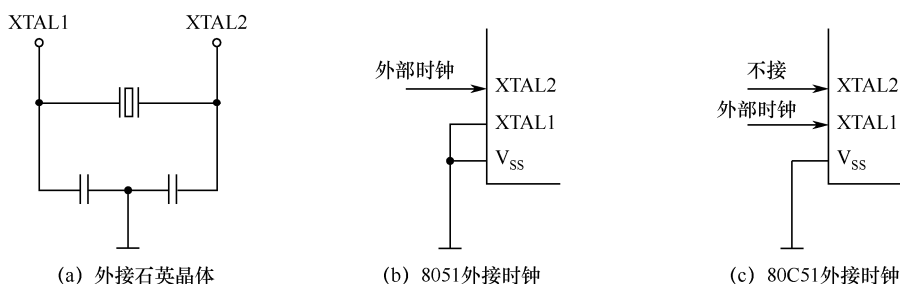


图1.4 8051单片机的时钟接法

V_{ss} (20): 接地。

V_{cc} (40): 接+5V电源。

\overline{EA}/V_{pp} (31): 访问外部存储器的控制信号。当 \overline{EA} 为高电平时,访问单片机片内程序存储器,当程序计数器PC的值超过0FFFH(对8051)或1FFFH(对8052)时,将自动转向执行单片机片外程序存储器内的程序。当 \overline{EA} 保持低电平时,只访问外部程序存储器,不管是否有片内程序存储器。第二功能 V_{pp} 为对8751片内EPROM的21伏编程电源输入。

$\overline{ALE}/\overline{PROG}$ (30): \overline{ALE} 是地址锁存允许信号,在访问外部存储器时,用来锁存由P0口送出的低8位地址信号。在不访问外部存储器时, \overline{ALE} 以振荡频率1/6的固定速率输出脉冲信号。因此它可用作对外输出的时钟。但要注意,只要外接有存储器,则 \overline{ALE} 端输出的就不再是连续的周期脉冲信号了。第二功能 \overline{PROG} 是用于对8751片内EPROM编程的脉冲输入端。

\overline{PSEN} (29): 外部程序存储器ROM的读选通信号。在执行访问外部ROM指令的时候,会自动产生 \overline{PSEN} 信号,而在访问外部数据存储器RAM或访问内部ROM时,不产生 \overline{PSEN} 信号。

P1.0~P1.7 (1~8): 双向I/O口P1。P1口能驱动(吸收或输出电流)4个LS型TTL负载。在对EPROM编程和程序验证时,接收低8位地址。在8052单片机中,P1.0还用作定时器2的计数触发输入端T2,P1.1还用作定时器2的外部控制端T2EX。

P3.0~P3.7 (10~17): 双向I/O口P3,P3口能驱动(吸收或输出电流)4个LS型TTL负载。P3口的每条引脚都有各自的第二功能。

P0.0~P0.7 (39-32): 双向I/O口P0。第二功能是在访问外部存储器时,可分时用作低8位地址和8位数据线,在对8751编程和校验时,用于数据的输入和输出。P0口能以吸收电流的方式驱动8个LS型TTL负载。

P2.0~P2.7 (21~28): 双向I/O口P2。P2口可以驱动(吸收或输出电流)4个LS型TTL负载。第二功能是在访问外部存储器时,输出高8位地址。在对EPROM编程和校验时,它接收高位地址。

1.2 8051单片机的存储器结构

图1.5为8051单片机的存储器结构图。在物理上,它有3个存储器空间:程序存储器

(CODE空间)、片内数据存储器 (IDATA和DATA空间)、片外数据存储器 (XDATA空间)。访问不同存储器空间时须采用不同的指令。

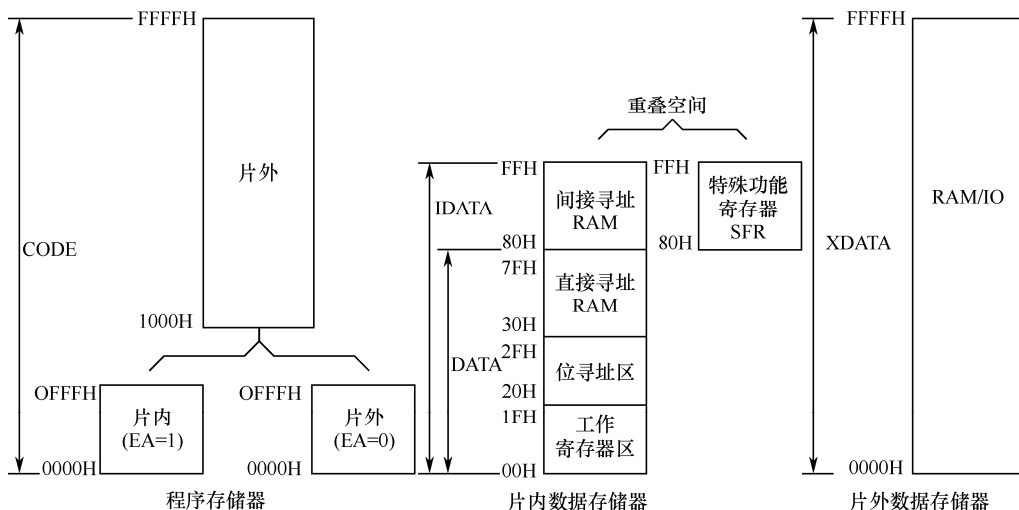


图1.5 8051单片机的存储器结构图

程序存储器ROM

8051单片机程序存储器ROM空间大小为64KB，地址范围为0000H~FFFFH，用于存放程序代码和一些表格常数，称为CODE空间。8051单片机专门提供一个引脚 \overline{EA} 来区分片内ROM和片外ROM。 \overline{EA} 引脚接高电平时，单片机从片内ROM中读取指令，当指令地址超过片内ROM空间范围后，就自动转向片外ROM读取指令； \overline{EA} 引脚接低电平时，所有的取指操作均对片外ROM进行。程序存储器的某些地址单元是保留给系统使用的：0000H~0002H单元是所有执行程序的入口地址，复位后，CPU总是从0000H地址开始执行程序；0003H~002BH单元均匀地分为5段，用于5个中断服务程序的入口，产生某个中断时，将自动进入其对应入口地址开始执行中断服务程序，一些新型8051单片机增加了更多的中断源，它们的中断入口地址也相应增加。

片外数据存储器RAM

8051单片机片外数据存储器RAM空间大小为64 KB，地址范围为0000H~FFFFH，称为XDATA空间。在XDATA空间内进行分页寻址操作时，称为PDATA区。

8051单片机采用所谓“哈佛”结构的存储器配置，即在物理上具有独立的ROM存储器和片外RAM数据存储器，而在逻辑上则采用相同的地址空间，其地址范围都是0000H~FFFFH，但是需要采用不同的指令和寻址方式来访问，从而可分别寻址64KB的ROM程序存储器和64KB的片外RAM数据存储器。

片内数据存储器RAM

8051单片机片内数据存储器RAM的空间最大为256 B，用于存放程序执行过程的各种变量及临时数据，整个片内RAM的地址范围00H~FFH被称为IDATA空间。片内RAM低128个字节（00H~7FH）被称为DATA空间。它既可用直接寻址访问，也可用间接寻址访问，而片内RAM高128个字节（80H~FFH）则只能采用间接寻址访问。片内RAM中00H~1FH地址范围被称为工作寄存器区，平均分为4组，每组都有8个工作寄存器R0~R7，在某一时刻，

CPU只能使用其中的一组工作寄存器，究竟选择哪一组工作寄存器，则由程序状态字寄存器PSW中RS0和RS1的状态决定，见表1.1。片内RAM中，20H~2FH地址范围被称为位寻址区（又称BDATA区）。其中每个存储器单元的每一位称为一个bit，可以用位处理指令直接操作。片内RAM中的位地址分配如图1.6所示。

RAM 地址	MSB								LSB
7FH									127
2FH	7E	7E	7D	7C	7B	7A	70	78	47
2FH	77	76	75	74	73	72	71	70	46
2DH	6F	6E	6D	6C	6B	6A	69	68	45
2CH	67	66	65	64	63	62	61	60	44
2BH	5F	5E	5D	5C	5B	5A	59	58	43
2AH	57	56	55	54	53	52	51	50	42
29H	4F	4E	4D	4C	4B	4A	49	48	41
28H	47	46	45	44	43	42	41	40	40
27H	3F	3E	3D	3C	3B	3A	39	38	39
26H	37	36	35	34	33	32	34	30	38
25H	2F	2E	2D	2C	2B	2A	29	28	37
24H	27	26	25	24	23	22	21	20	36
23H	1F	1E	1D	1C	1B	1A	19	18	35
22H	17	16	15	14	13	12	11	10	34
21H	0F	0E	0D	0C	0B	0A	09	08	33
20H	07	06	05	04	03	02	01	00	32
1FH	工作寄存器3区								31
18H									24
17H	工作寄存器2区								23
10H									16
0FH	工作寄存器1区								15
08H									8
07H	工作寄存器0区								7
00H									0

图1.6 片内RAM中的位地址分配

8051单片机在与IDATA空间高128个字节（80H~FFH地址范围）安排了一个重叠空间被称为特殊功能寄存器区（又称SFR区），地址也为80H~FFH，但在使用时，可通过指令加以区别。有些特殊功能寄存器是可以位寻址的，其可寻址位被称为sbit。表1.2为8051单片机特殊功能寄存器地址及符号表。表中带*号的为可位寻址的特殊功能寄存器。片内RAM中的各个单元都可以通过其地址来寻找。对于工作寄存器，一般使用R0~R7表示，对于特殊功能寄存器，也是直接用其符号名较为方便。需要指出的是，8051单片机的堆栈必须使用片内RAM，而片内RAM空间十分有限，因此要仔细安排堆栈指针SP的值，以保证不会发生堆栈溢出而导致系统崩溃。

表1.2 8051单片机特殊功能寄存器地址及符号表

特殊功能寄存器	片内 RAM 地址	说 明
*ACC	E0H	累加器
*B	F0H	乘法寄存器
*PSW	D0H	程序状态字寄存器
SP	81H	堆栈指针
DPL	82H	数据指针（低 8 位）
DPH	83H	数据指针（高 8 位）
*IE	A8H	中断允许寄存器
*IP	B8H	中断优先级寄存器
*P0	80H	P0 口锁存器
*P1	90H	P1 口锁存器
*P2	A0H	P2 口锁存器
*P3	B0H	P3 口锁存器
PCON	87H	电源控制及波特率选择寄存器
*SCON	98H	串行口控制寄存器
SBUF	99H	串行数据缓冲器
*TCON	88H	定时器控制寄存器
TMOD	89H	定时器方式选择寄存器
TL0	8AH	定时器 0 低 8 位
TH0	8BH	定时器 0 高 8 位
TL1	8CH	定时器 1 低 8 位
TH1	8DH	定时器 1 高 8 位

1.3 CPU时序

8051单片机内部有一个高增益反向放大器，用于构成振荡器，反向放大器的输入端为XTAL1，输出端为XTAL2，分别是8051的19脚和18脚。在XTAL1和XTAL2之间接一个石英晶体及两个电容就可以构成稳定的自激振荡器，如图1.7所示。晶体振荡器的振荡信号经过片内时钟发生器进行2分频，向CPU提供两相时钟信号P1和P2。时钟信号的周期被称为状态时间S。它是振荡周期的2倍，在每个状态的前半周期P1信号有效，在每个状态的后半周期P2信号有效，CPU就以这两相时钟信号为基本节拍指挥单片机各部分协调工作。

CPU执行一条指令所需要的时间是以机器周期为单位的。8051单片机的一个机器周期包括12个振荡周期，分为6个S状态：S1~S6。每个状态又分为2拍，即前面介绍的P1和P2信号。因此一个机器周期中的12个振荡周期可表示为S1P1，S1P2，S2P1，…，S6P1，S6P2。当采用12MHz的晶体振荡器时，一个机器周期为1μs。CPU执行一条指令通常需要1~4个机器周期，指令的执行速度与其需要的机器周期数直接有关，所需机器周期数越少，速度越快，8051单片机只有乘、除两条指令，需要4个机器周期，其余均为单周期或双周期指令。

图1.8为几种典型的取指令和执行时序。从图中可以看到，在每个机器周期之内，地址锁存信号ALE两次有效，第一次出现在S1P2和S2P1期间，第二次出现在S4P2和S5P1期间。

单周期指令的执行从S1P2开始，此时操作码被锁存在指令寄存器内。若是双字节指令，则在同一机器周期的S4状态读第2个字节。若是单字节指令，则在S4状态仍进行读，但操作无效，且程序计数器PC的值不加1。

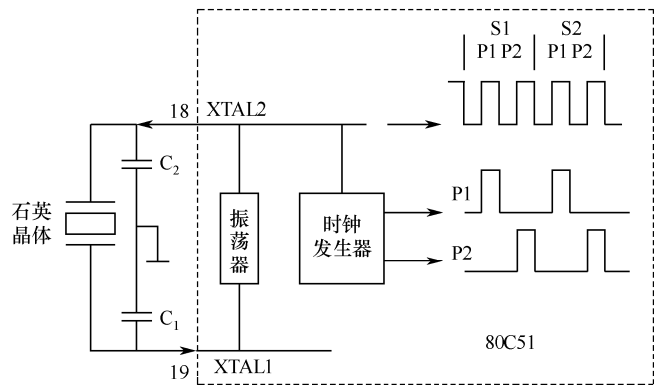


图1.7 8051片内振荡器及时钟发生电路

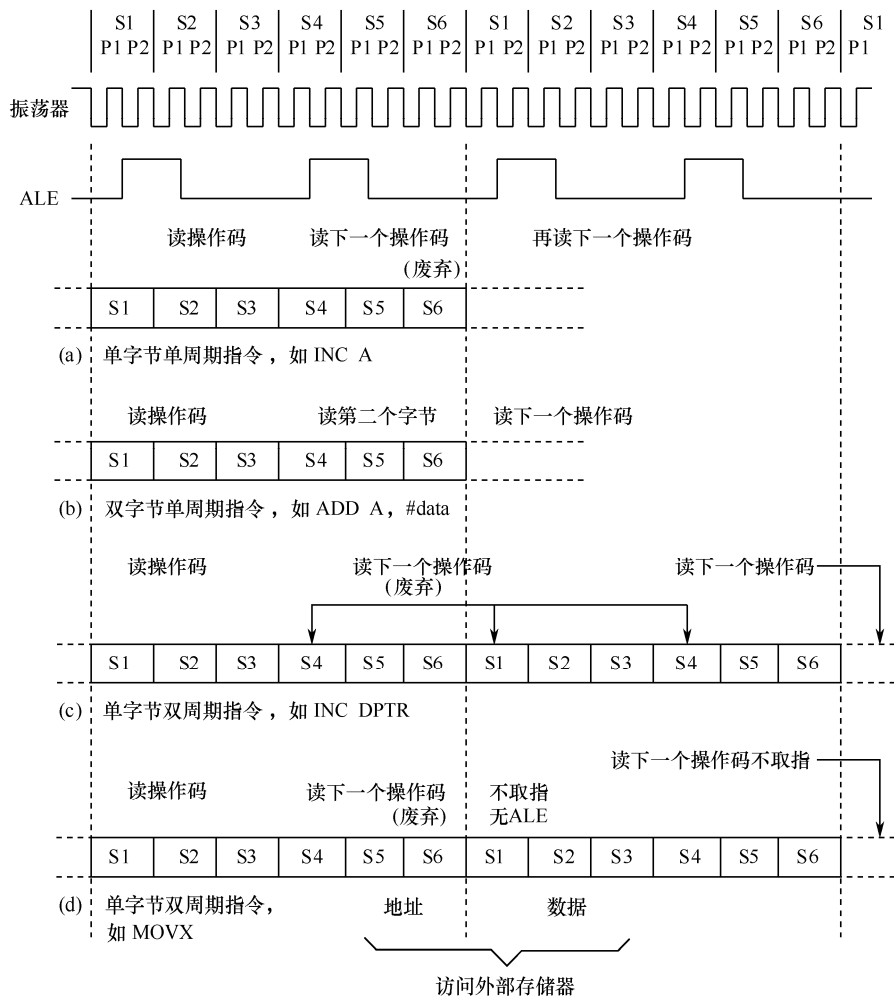


图1.8 几种典型的取指和执行周期时序

图1.8 (a) 和图1.8 (b) 分别为单字节单周期和双字节单周期指令的时序，它们都在S6P2结束时完成操作。

图1.8 (c) 为单字节双周期指令的时序，在两个机器周期内进行4次操作，由于是单字节指令，所以后面的3次操作无效。

图1.8 (d) 为CPU访问片外数据存储器指令“MOVX”的时序，是一条单字节双周期指令，在第一个机器周期的S5状态开始送出片外数据存储器的地址进行数据的读/写操作。在此期间没有ALE信号，所以在第二个周期不会产生取指操作。

1.4 复位信号与复位电路

8051单片机与其他微处理器一样，在启动时需要复位，使CPU和系统的各个部件都处于一种确定的初始状态。复位信号从单片机的RST引脚输入，高电平有效，其有效电平应维持至少两个机器周期，若采用6MHz的晶体振荡器，则复位信号至少应持续4 μ s以上，才可以保证可靠复位。

复位操作有上电自动复位和按键手动复位两种方式。上电自动复位是通过外部复位电路的电容充电来实现的。其电路如图1.9 (a) 所示。只要电源Vcc电压上升时间不超过1ms，通过在Vcc和RST之间加一个22 μ F的电容，RST和Vss引脚（即地）之间加一个1k Ω 的电阻，就可以实现上电自动复位。

按键手动复位电路如图1.9 (b) 所示。它是在上电自动复位电路的基础上增加一个电阻R1和一个按键RESET实现的。它不仅具有上电自动复位功能，而且在按下RESET按钮后，电容C通过R1放电，同时电源Vcc通过R1和R2分压，而R2的阻值要比R1的阻值大很多，大部分电压都降落在R2上，从而使RST端得到一个高电平，导致单片机复位。

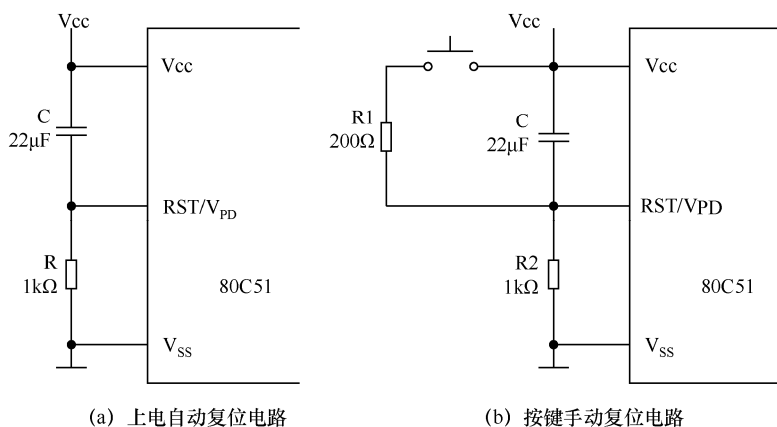


图1.9 复位电路

上述电路中的电阻、电容参数适用于6MHz的外接晶振，能保证复位信号持续两个机器周期的高电平。复位电路虽然简单，但其作用非常重要，一个实际单片机应用系统能否正常工作，首先要检查能否产生正确的复位信号。复位以后，单片机内部各寄存器的状态见表1.3。复位不影响片内RAM的内容。

表1.3 单片机内部各寄存器的状态

寄 存 器	状 态	寄 存 器	状 态
PC	0000H	TMOD	00H
ACC	00H	TL0	00H
PSW	00H	TH0	00H
SP	07H	TL1	00H
DPTR	0000H	TH1	00H
P0~P3	FFH	SCON	00H
IP	××000000B	SBUF	不定
IE	0×000000B	PCON	0×××0000B

1.5 并行I/O端口结构

8051单片机有四个并行I/O端口，称其为P0、P1、P2、P3。每个端口都有8根引脚，共有32根I/O引脚。它们都是双向通道，每一条I/O引脚都能独立地用作输入或输出，作为输出时数据可以锁存，作为输入时数据可以缓冲。

图1.10为P0~P3各口中的一位逻辑图。P0~P3口各有一个锁存器，分别对应4个特殊功能寄存器地址：80H、90H、A0H、B0H。这四个I/O口的功能不完全相同，负载能力也不相同，P1、P2、P3都能驱动四个LSTTL门电路，并且不需外加电阻就能直接驱动MOS电路。P0口在驱动TTL电路时能带动8个LS型TTL门电路，若作为地址/数据总线，则可直接驱动；而作为I/O口时，则需外接上拉电阻才能驱动MOS电路。

P0为三态双向口，可作为输入、输出端口使用，也可作为系统扩展时的低8位地址/8位数据总线使用。P0口内部有一个2选1的MUX开关，当8051以单芯片方式工作而不需要外部扩展时，内部控制信号将使MUX开关接通到锁存器，此时P0口作为双向I/O端口，由于P0口没有内部上拉电阻，通常要在外部加一个上拉电阻来提高驱动能力。当8051需要进行外部扩展时，内部控制信号将使MUX开关接通到内部地址/数据线，此时P0口在ALE信号的控制下分时输出低8位地址和8位数据信号。

P1口为准双向口，它的每一位都可以分别定义为输入或输出使用。P1口作为输入口使用时有两种工作方式，即所谓“读端口”和“读引脚”。读端口时，实际上并不从外部读入数据，而只把端口锁存器中的内容读入到内部总线，经过某种运算和变换后，再写回到端口锁存器，属于这类操作的指令很多，如对端口内容取反等。读引脚时才真正地把外部的输入信号读入到内部总线。逻辑图中各有两个输入缓冲器，CPU根据不同的指令分别发出“读端口”或“读引脚”信号，以完成两种不同的操作。在读引脚，也就是从外部输入数据时，为了保证输入正确的外部输入电平信号，首先要向端口锁存器写入一个“1”，再进行读引脚操作，否则，端口锁存器中原来状态有可能为“0”，加到输出驱动场效应管栅极的信号为“1”，该场效应管导通，对地呈现低阻抗。这时即使引脚上输入的是“1”信号，也会因端口的低阻抗而使信号变化，使得外加的“1”信号写入时不一定是“1”。若先执行置“1”操作，则可使驱动场效应管截止，引脚信号直接加到三态缓冲器，实现正确的写入。正是由于P1口在进行输入操作之前需要有这样一个附加准备动作，故称之为“准双向口”。P1作为输

出口时,如果要输出“1”,则只要将“1”写入P1口的某一位锁存器,使输出驱动场效应管截止。该位的输出引脚由内部上拉电阻拉成高电平,即输出为“1”。要输出“0”时,将“0”写入P1口的某一位锁存器,使输出驱动场效应管导通,该位的输出引脚被接到地端,即输出为“0”。

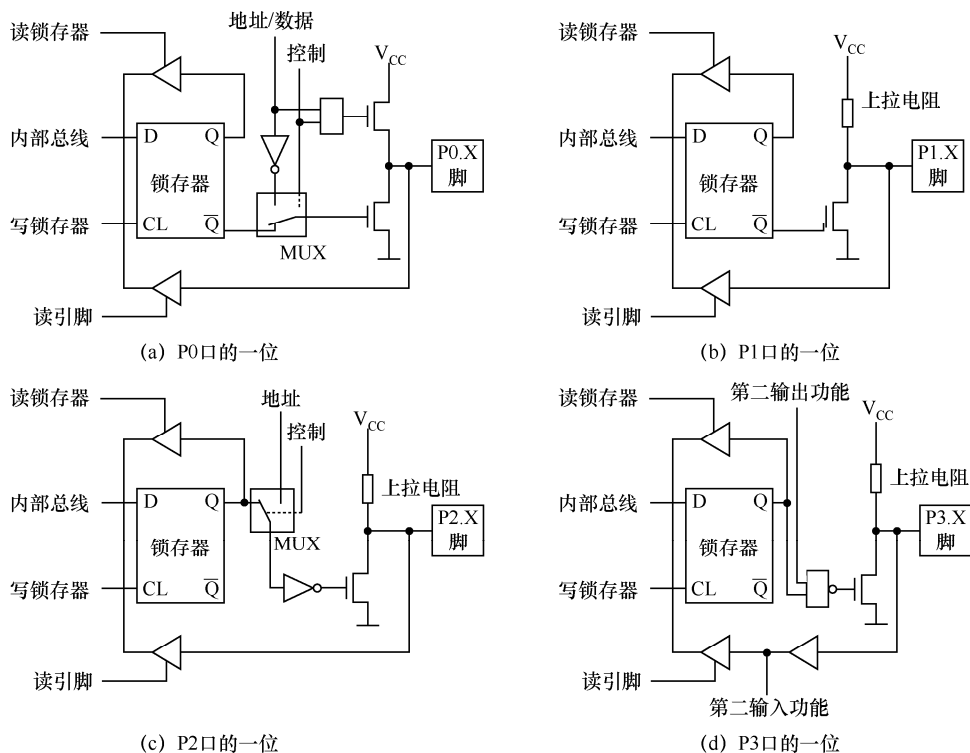


图1.10 P0~P3各口中的一位逻辑图

P2口也是一个准双向口,有两种使用功能:作为普通I/O端口或作为系统扩展时的高8位地址总线。P2口内部结构与P0口类似,也有一个2选1的MUX开关,P2口作为I/O端口使用时,内部控制信号使MUX开关接通到锁存器,此时P2口的用法与P1口相同。P2口作为外部地址总线使用时,内部控制信号使MUX开关接通到内部地址线,此时P2口的引脚状态由所输出的地址决定。需要特别指出的是,只要进行了外部系统扩展,则由于对片外地址的操作是连续不断的,故此时P0口和P2口就不能再用作I/O端口了。

P3口为多功能口,除了用作通用I/O端口之外,它的每一位都有各自的第二功能,见表1.4。P3口作为通用I/O端口时,其使用方法与P1口相同,P3口的第二功能可以单独使用,即不用第二功能的引脚仍可以作为通用I/O口线使用。

表1.4 P3口的第二功能定义

端口引脚	第二功能	端口引脚	第二功能
P3.0	RXD (串行输入口)	P3.4	T0 (定时器0外部输入)
P3.1	TXD (串行输出口)	P3.5	T1 (定时器1外部输入)
P3.2	$\overline{\text{INT0}}$ (外部中断0输入)	P3.6	$\overline{\text{WR}}$ (外部RAM写选通)
P3.3	$\overline{\text{INT1}}$ (外部中断1入)	P3.7	$\overline{\text{RD}}$ (外部RAM读选通)

8051单片机没有独立的对外地址、数据和控制“三总线”，当需要进行外部扩展时，需要采用I/O端口的复用功能，将P0、P2口用作地址/数据总线，P3口用其第二功能，形成外部地址、数据和控制总线，如图1.11所示。

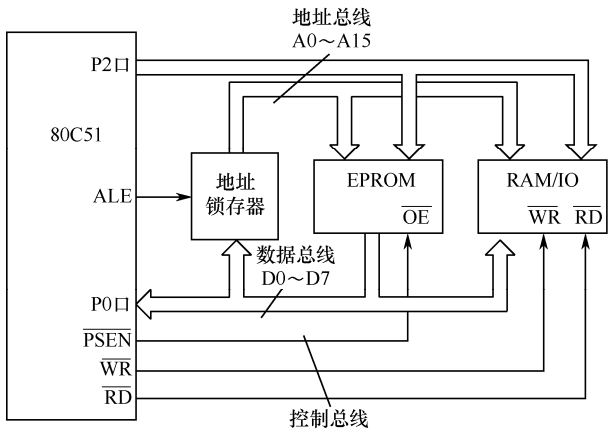


图1.11 单片机与外部存储器、I/O端口的连接

P0口在进行外部扩展时分时复用，在读/写片外存储器时，P0口先送出低8位地址信号。该信号只能维持很短的时间，然后P0口又送出8位数据信号。为了在整个读/写片外存储器期间都存在有效的低8位地址信号，必须在P0口上外接一个地址锁存器，在ALE信号有效期间将低8位地址锁存于锁存器内，再从这个锁存器对外输出低8位地址。P2口在进行外部扩展时只用作高8位地址线，在整个读写期间，P2口输出信号维持不变，因此P2口不需外接锁存器。一般在片外接有存储器时，P0和P2口不能再用作通用I/O端口，此时只有P1口可作为通用I/O端口，P3口没有使用第二功能的引脚还可以用作I/O端口线。另外还要注意，外接程序存储器ROM的读写选通信号为PSEN，而外接数据存储器RAM的读/写选通信号为RD和WR，从而保证外部ROM和外部RAM不会发生混淆。

1.6 指令系统

指令系统是一套控制单片机执行操作的编码，是单片机能直接识别的命令。指令系统在很大程度上决定了单片机的功能和使用是否方便灵活。指令系统对于用户来说也是十分重要的，只有详细了解了单片机的指令功能才能编写出高效的软件程序。指令本身是一组二进制数代码，记忆起来很不方便，为了便于记忆，将这些代码用具有一定含义的指令助记符来表示。助记符一般采用有关英文单词的缩写，这样就容易理解和记忆单片机的各种指令了。下面是两条分别用代码形式和助记符形式书写的指令：

十六进制代码	助记符	功能
740A	MOV A, #0AH	； 将十六进制数0AH放入累加器A中
2414	ADD A, #14H	； 累加器A中的内容与十六进制数14H
		； 相加，结果放在累加器A中

尽管采用助记符后，书写的字符增多了，但由于增强了可读性，使用时会觉得更方便。

采用助记符和其他一些符号来编写的指令程序被称为汇编语言源程序。汇编语言源程序经过汇编之后即可得到可执行的机器代码目标程序。

一条指令通常由两部分组成：操作码和操作数。操作码用来规定这条指令完成什么操作，如是做加减运算，还是数据传送等。操作数表示这条指令所完成的操作对象，即是对谁进行操作。操作数可以直接是一个数，或者是一个数所在的内存地址。

操作码和操作数都是二进制代码。在8051单片机中，8位二进制数为一个字节，指令是由指令字节组成的。对于不同的指令，指令的字节数不相同。8051单片机有单字节、双字节或三字节指令。

单字节指令中既包含操作码的信息，也包含操作数的信息。这可能有两种情况。一种是指令的含义和对象都很明确，不必再用另一个字节来表示操作数。例如，数据指针加1指令：INC DPTR，由于操作的内容和对象都很明确，故不必再加操作数字节，其指令码为

10100011

另一种情况是用一个字节中的几位来表示操作数或操作数所在的位置。例如，从工作寄存器向累加器A传送数据的指令：MOV A, Rn。其中，Rn可以是8个工作寄存器R0~R7中的一个，在指令码中分出三位来表示这8个工作寄存器，用其余各位表示操作码的作用，指令码为

11101rrr

其中，最低3位码用来表示从哪个寄存器取数，故一个字节也就够了。8051单片机共有49条单字节指令。

双字节指令一般是用一个字节表示操作码，另一个字节表示操作数或操作数的地址。这时操作数或其地址就是一个8位的二进制数，因此必须专门用一个字节来表示。例如，8位二进制数传送到累加器A的指令：MOV A, #data。其中，#data表示8位二进制数，也叫立即数，这就是双字节指令，其指令码为

01110100

#data

双字节指令的第二个字节也可以是操作数所在的地址。8051单片机共有45条双字节指令。

三字节指令则是一个字节的操作码、两个字节的操作数。操作数可以是数据，也可以是地址，因此可能有如下四种情况：

操作码	立即数	立即数
操作码	地址	立即数
操作码	立即数	地址
操作码	地址	地址

8051单片机共有17条三字节指令，只占全部指令的15%。一般而言，指令的字节数越少，执行速度越快。从这个角度来说，8051单片机的指令系统是比较合理的。

1.7 指令的寻址方式

所谓寻址，就是寻找操作数的地址。在用汇编语言编程时，数据的存放、传送、运算都要通过指令来完成。编程者必须自始至终十分清楚操作数的位置，以及如何将它们传送到适当的寄存器去运算。因此，如何从各个存放操作数的区域去寻找和提取操作数就变得十分重要。所谓寻址方式就是通过确定操作数所在的地址把操作数提取出来的方法。

8051单片机有7种寻址方式，分别说明如下。

1.7.1 寄存器寻址

寄存器寻址就是以通用寄存器的内容作为操作数，在指令的助记符中直接以寄存器的名字来表示操作数的位置。8051单片机没有专门的通用硬件寄存器，而是把内部数据RAM区中00~1FH地址单元作为工作寄存器使用，共有32个地址单元，分成4组，每组8个工作寄存器，命名为R0~R7，每次可以使用其中一组。当以R0~R7来表示操作数时，就属于寄存器寻址方式，如

```
MOV A, R0
ADD A, R0
```

前一条指令是将R0寄存器的内容传送到累加器A中，后一条指令则是对A和R0的内容做加法运算。

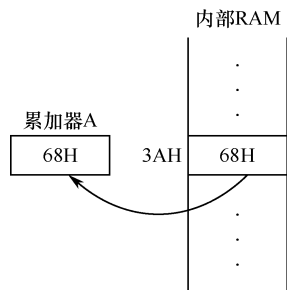
特殊功能寄存器B也可当作通用寄存器使用，但用B表示操作数地址的指令不属于寄存器寻址，而是属于下面所讲的直接寻址。

1.7.2 直接寻址

在指令中直接给出操作数地址就属于直接寻址方式。在这种方式中，指令的操作数部分直接是操作数地址。在8051单片机中，用直接寻址方式可以访问片内数据RAM中DATA空间的00H~7FH共128个字节单元及所有的特殊功能寄存器。在指令助记符中，直接寻址的地址可用两位十六进制数表示。对于特殊功能寄存器，还可用它们各自的名称符号来表示，以增加程序的可读性，如

```
MOV A, 3AH
```

就属于直接寻址。其中，3AH所表示的就是内部RAM地址。这条指令的功能是将内部RAM中3AH字节单元的内容传送到累加器A。该指令的功能如图1.12所示。



1.7.3 立即寻址

若指令的操作数是一个8位或16位二进制数，就称为立即寻

图1.12 直接寻址指令的功能

址。指令中的操作数被称为立即操作数。由于8位立即数和直接地址都是8位二进制数（或两位16进制数），故为区分起见，在立即数前面冠以“#”号，如#3AH表示立即数3AH，直接写3AH则表示RAM区中地址为3AH的字节单元，如指令

```
MOV A, #3AH
MOV A, 3AH
```

前一条指令为立即寻址，执行后，累加器A中的内容变为3AH；后一条指令为直接寻址，执行后，累加器A中的内容变为RAM区中地址为3AH字节单元的内容。8051单片机只有一条16位立即数指令：

```
MOV DPTR, #data16
```

其功能是将16位立即数送往数据指针寄存器。由于是16位立即数，需要用两个字节表示，因此这是一条三字节的指令，即一字节指令码、二字节立即数，指令格式为：

1 0 0 1 0 0 0 0

立即数高 8 位

立即数低 8 位

1.7.4 寄存器间接寻址

若以寄存器的名称间接给出操作数的地址，则称为寄存器间接寻址。在这种寻址方式下，指令中工作寄存器的内容不是操作数，而是操作数的地址。指令执行时，先通过工作寄存器的内容取得操作数地址，再到此地址所规定的存储单元取得操作数。

8051单片机可采用寄存器间接寻址方式访问全部256个字节的片内RAM地址单元00H~FFH（即IDATA空间），也可访问64KB的外部RAM（即XDATA空间），但是这种寻址方式不能访问特殊功能寄存器。只能采用工作寄存器R0、R1或数据指针寄存器DPTR来进行间接寻址，在寄存器R0、R1或DPTR名称前面加一个符号@来表示寄存器间接寻址，如

```
MOV A, @R0
```

该指令的功能如图1.13所示。指令执行之前，R0寄存器的内容3AH是操作数的地址，内部RAM中地址为3AH单元的内容65H才是操作数，执行后，累加器A中的内容变为65H。若采用寄存器寻址指令

```
MOV A, R0
```

则执行后，累加器A中的内容变为3AH，对这两类指令的差别和用法一定要区分清楚，正确使用。

1.7.5 变址寻址

变址寻址是以某个寄存器的内容为基本地址，然后在这个基址上加上一定的偏移量后才是真正的操作数地址。8051单片机没有专门的变址寄存器，而是采用数据指针寄存器DPTR或程序计数器指针PC SP的内容为基本地址。地址偏移量则是累加器A中的内容，将基址与偏移量相加，即以DPTR或者PC的内容与A的内容之和作为实际的操作数地址。8051单片机采用变址寻址方式可以访问64KB的ROM程序存储器（即CODE空间），如指令

MOVC A, @A+DPTR

该指令的功能如图1.14所示。指令执行前 (A) =11H, (DPTR) =02F1H, 故实际操作数的地址应为02F1H+11H=0302H。指令执行后, 将程序存储器ROM中0302H单元的内容1EH传送到累加器A。需要注意的是, 虽然在变址寻址时采用数据指针DPTR作为基址寄存器, 但变址寻址的区域都是程序存储器ROM而不是数据存储器RAM。另外, 尽管变址寻址方式的指令助记符和指令操作都较为复杂, 但却是一字节指令。

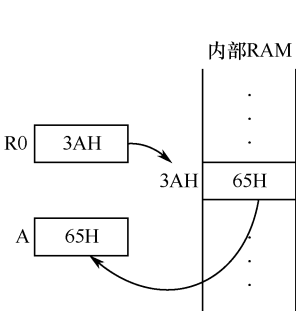


图1.13 寄存器间接寻址指令的功能

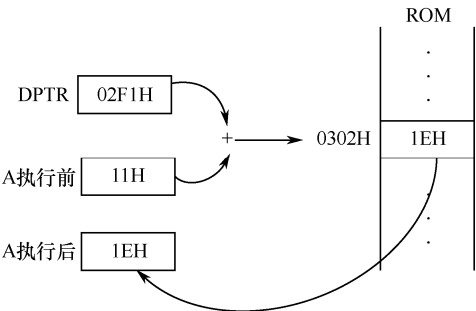


图1.14 变址寻址指令的功能

1.7.6 相对寻址

8051单片机设有直接转移指令和相对转移指令。相对转移指令需要采用相对寻址方式, 此时指令的操作数部分给出的是地址的相对偏移量。在指令中以rel表示相对偏移量, rel为一个带符号的常数, 可正也可以负。若rel值为负数, 则应用补码表示。一般将相对转移指令本身所在的地址称为源地址, 转移后的地址称为目的地址。它们的关系为

目的地址 = 源地址 + 指令字节数 + rel

如指令

SJMP rel

该指令的功能如图1.15所示。这条指令的机器码为80H, rel, 共两个字节。设该指令所在的源地址为2000H, rel的值为54H, 则转移后的目的地址为2000H+02+54H=2056H。

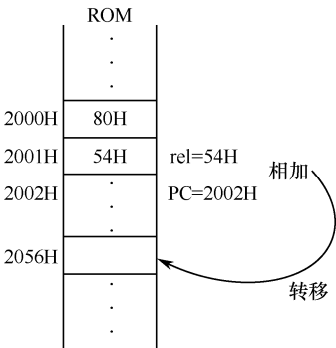


图1.15 相对寻址指令的功能

1.7.7 位寻址

采用位寻址方式的指令，其操作数是8位二进制数中的某一位，在指令中要给出位地址，位地址可以是片内RAM中可位寻址区20H~2FH（即BDATA区）某个字节单元的某一位，或者是可位寻址特殊功能寄存器的某一位。表1.5列出了可位寻址特殊功能寄存器及其位地址。

表1.5 可位寻址特殊功能寄存器及其位地址

特殊功能寄存器	单元地址	表示符号	位地址
P0	80H	P0.0~P0.7	80H~87H
TCON	88H	TCON.0~TCON.7	88H~8FH
P1	90H	P1.0~P1.7	90H~97H
SCON	98H	SCON.0~SCON.7	98H~9FH
P2	A0H	P2.0~P2.7	A0H~A7H
IE	A8H	IE.0~IE.7	A8H~AFH
P3	B0H	P3.0~P3.7	B0H~B7H
IP	B8H	IP.0~IP.7	B8H~BFH
PSW	D0H	PSW.0~PSW.7	D0H~D7H
ACC	E0H	ACC.0~ACC.7	E0H~E7H
B	F0	B.0~B.7	F0H~F7H

位地址可采用以下几种方式表示：

- ① 直接用位地址00H~FFH来表示，如20H字节单元的0~7位可表示为20H~27H；
- ② 采用第*n*字节单元第*n*位来表示，如25H.5表示25H字节单元的第5位；
- ③ 对于特殊功能寄存器可以用寄存器名加位数的表示方法，如PSW.7，也可以直接用其特殊功能位名称，如CY；
- ④ 用汇编语言中的伪指令定义。

如指令

```
SETB PSW.7
SETB CY
```

这两条指令具有相同的功能，都是将程序状态字PSW的最高位置1。

1.8 指令分类详解

8051单片机共有111条指令，按指令功能可分为算术运算指令、逻辑运算指令、数据传送指令、控制转移指令及位操作指令5大类。

1.8.1 算术运算指令

算术运算指令包括加、减、乘、除法指令。加法指令又分为普通加法指令、带进位加法

指令和加1指令。

1. 普通加法指令

ADD A, Rn	;Rn (n=0~7) 为工作寄存器
ADD A, direct	;direct为直接地址单元
ADD A, @Ri	;Ri (i=0~1) 为工作寄存器
ADD A, #data	;#data为立即数

这组指令的功能是将累加器A的内容与第二操作数的内容相加，结果送回到累加器A中。在执行加法的过程中，如果位7有进位，则置“1”进位标志CY，否则清“0”CY。如果位3有进位，则置“1”辅助进位标志AC，否则清“0”AC。如果位6有进位而位7没有进位，或者位7有进位而位6没有进位，则置“1”溢出标志OV，否则清“0”OV。

2. 带进位加法指令

ADDC A, Rn	;Rn (n=0~7) 为工作寄存器
ADDC A, direct	;direct为直接地址单元
ADDC A, @Ri	;Ri (i=0~1) 为工作寄存器
ADDC A, #data	;#data为立即数

这组指令的功能与普通加法指令类似，唯一的不同之处是在执行加法时，还要将上一次进位标志CY的内容也一起加进去，对于标志位的影响与普通加法指令相同。

3. 加1指令

INC A	
INC Rn	;Rn (n=0~7) 为工作寄存器
INC direct	;direct为直接地址单元
INC @Ri	;Ri (i=0~1) 为工作寄存器
INC DPTR	;DPTR为16位数据指针寄存器

这组指令的功能是将所指出操作数的内容加1。如果原来的内容为0FFH，则加1后将产生上溢出，使操作数的内容变成00H，但不影响任何标志。指令 INC DPTR是对16位的数据指针寄存器DPTR执行加1操作，指令执行时，先对数据指针的低8位DPL的内容加1，当产生上溢出时，就对数据指针的高8位DPH加1，但不影响任何标志。

4. 十进制调整

DA A

这条指令的功能是对累加器A中的内容进行BCD码调整，通常用于BCD码运算程序中，使A中的运算结果为两位BCD码数。

5. 带进位减法指令

SUBB A, Rn	;Rn (n=0~7) 为工作寄存器
SUBB A, direct	;direct为直接地址单元
SUBB A, @Ri	;Ri (i=0~1) 为工作寄存器
SUBB A, #data	;#data为立即数

这组指令的功能是将累加器A的内容与第二操作数的内容相减，同时还要减去上一次进位标志CY的内容，结果送回到累加器A中。在执行减法的过程中，如果位7有借位，则置“1”当前进位标志CY，否则清“0”CY。如果位3有借位，则置“1”辅助进位标志AC，否则清“0”AC。如果位6有借位而位7没有借位，或者位7有借位而位6没有借位，则置“1”溢出标志OV，否则清“0”OV。

6. 减1指令

```
DEC A
DEC Rn          ;Rn (n=0~7) 为工作寄存器
DEC direct      ;direct为直接地址单元
DEC @Ri         ;Ri (i=0~1) 为工作寄存器
```

这组指令的功能是将所指出操作数的内容减1，如果原来的内容为00H，则减1后将产生下溢出，使操作数的内容变成0FFH，但不影响任何标志。

7. 单字节乘法指令

```
MUL AB
```

这条指令的功能是将累加器A中的8位无符号整数与寄存器B中的8位无符号整数相乘，乘积为16位整数。乘积的低8位存放在累加器A中，高8位存放在寄存器B中。如果乘积大于255 (0FFH)，则置“1”溢出标志OV，否则清“0”OV。进位标志总是被清“0”。

8. 单字节除法指令

```
DIV AB
```

这条指令的功能是将累加器A中的8位无符号整数除以寄存器B中的8位无符号整数，所得商的整数部分存放在累加器A中，余数部分存放在寄存器B中，清“0”进位标志CY和溢出标志OV。如果原来B中的内容为0（被0除），则执行除法后，A和B中的内容不定，并置“1”溢出标志OV，在任何情况下，进位标志总是被清“0”。

1.8.2 逻辑运算指令

逻辑运算指令分为简单逻辑操作指令、逻辑与指令、逻辑或指令及逻辑异或指令。

1. 简单逻辑指令

```
CLR A          ;对累加器A清"0"
CPL A          ;对累加器A的内容求反
RL A           ;累加器A的内容向左环移一位
RLC A          ;累加器A的内容带进位位CY向左环移一位
RR A           ;累加器A的内容向右环移一位
RRC A          ;累加器A的内容带进位位CY向右环移一位
SWAP A         ;将累加器A的高半字节 (A.7~A.4) 与低半字节 (A.3~A.0) 交换
```

这组指令的功能是直接对累加器A的内容进行简单逻辑操作，结果仍在累加器A中。

2. 逻辑与指令

ANL A, Rn	; (A) \wedge (Rn) \rightarrow A, n=0~7
ANL A, direct	; (A) \wedge (direct) \rightarrow A
ANL A, @Ri	; (A) \wedge ((Ri)) \rightarrow A, i=0或1
ANL A, #data	; (A) \wedge #data \rightarrow A
ANL direct, A	; (direct) \wedge (A) \rightarrow direct
ANL direct, #data	; (direct) \wedge #data \rightarrow direct

这组指令的功能是将两个操作数的内容按位进行逻辑与运算，结果送入累加器A 或由direct所指出的内部RAM单元。

3. 逻辑或指令

ORL A, Rn	; (A) \vee (Rn) \rightarrow A, n=0~7
ORL A, direct	; (A) \vee (direct) \rightarrow A
ORL A, @Ri	; (A) \vee ((Ri)) \rightarrow A, i=0或1
ORL A, #data	; (A) \vee #data \rightarrow A
ORL direct, A	; (direct) \vee (A) \rightarrow direct
ORL direct, #data	; (direct) \vee #data \rightarrow direct

这组指令的功能是将两个操作数的内容按位进行逻辑或运算，结果送入累加器A 或由direct所指出的内部RAM单元。

4. 逻辑异或指令

XRL A, Rn	; (A) \oplus (Rn) \rightarrow A, n=0~7
XRL A, direct	; (A) \oplus (direct) \rightarrow A
XRL A, @Ri	; (A) \oplus ((Ri)) \rightarrow A, i=0或1
XRL A, #data	; (A) \oplus #data \rightarrow A
XRL direct, A	; (direct) \oplus (A) \rightarrow direct
XRL direct, #data	; (direct) \oplus #data \rightarrow direct

这组指令的功能是将两个操作数的内容按位进行逻辑异或运算，结果送入累加器A 或由direct所指出的内部RAM单元。

1.8.3 数据传送指令

8051单片机的存储器空间可分为如下3个部分，即

ROM存储器（CODE空间）：0000H~FFFFH

片内RAM存储器（IDATA空间）：00H~FFH

片外RAM存储器/扩展IO端口（XDATA空间）：0000H~FFFFH

指令对哪一个存储器空间进行操作是由指令的操作码和寻址方式确定的。对于程序存储器ROM只能通过变址寻址方式采用MOVC指令访问，对于特殊功能寄存器只能采用直接寻址和位寻址方式，不能采用间接寻址方式；对于8051单片机片内RAM的高128字节则只能采用寄存器间接寻址方式，而片内RAM的低128个字节则既能间接寻址，也能直接寻址；片外RAM存储器/扩展IO端口只能通过间接寻址方式用MOVX指令访问。

1. 数据传送到累加器A的指令

```
MOV A, Rn          ;n=0~7
MOV A, direct
MOV A, @Ri         ;i=0或1
MOV A, #data
```

这组指令的功能是把源操作数的内容送入累加器A。

2. 数据传送到工作寄存器Rn的指令

```
MOV Rn, A          ;n=0~7
MOV Rn, direct     ;n=0~7
MOV Rn, #data      ;n=0~7
```

这组指令的功能是把源操作数的内容送入当前工作寄存器区中的某一个寄存器R0~R7。

3. 数据传送到内部RAM单元或特殊功能寄存器SFR的指令

```
MOV direct, A
MOV direct, Rn     ;n=0~7
MOV direct, direct
MOV direct, @Ri    ;i=0或1
MOV direct, #data
MOV @Ri, A         ;i=0或1
MOV @Ri, direct    ;i=0或1
MOV @Ri, #data     ;i=0或1
MOV DPTR, #data16
```

这组指令的功能是把源操作数的内容送入指定的片内RAM单元或特殊功能寄存器。最后一条指令的功能是将16位数据送入数据指针寄存器DPTR。

4. 堆栈操作指令

```
PUSH direct        ;进栈
POP direct          ;出栈
```

在8051单片机的特殊功能寄存器中有一个堆栈指针寄存器SP，进栈指令的功能是首先将堆栈指针SP的内容加1，然后将直接地址所指出的内容送入SP指出的内部RAM单元。出栈指令的功能是将SP所指出的内部RAM单元的内容送入由直接地址所指出的字节单元，同时将堆栈指针SP的内容减1。

5. 累加器A与外部数据存储器之间的数据传送指令

```
MOVX A, @DPTR      ;((DPTR))→A
MOVX A, @Ri        ;((P2Ri))→A, i=0或1
MOVX @DPTR, A      ;(A)→(DPTR)
MOVX @Ri, A        ;(A)→(P2Ri)
```

这组指令的功能是在累加器A与片外数据存储器RAM或扩展I/O端口之间进行数据

传送。

6. 查表指令

```
MOVC A, @A+PC
MOVC A, @A+DPTR
```

这是两条很有用的查表指令，它们可用来查找存放在程序存储器中的常数表格。其中，第一条指令是以程序计数器PC作为基址寄存器，累加器A的内容作为无符号数偏移量与PC的内容（下一条指令的起始地址）相加，得到一个16位的地址，并将该地址指出程序存储器单元的内容送入累加器A。这条指令的优点是不改变特殊功能寄存器和PC的状态，只要根据A中的内容就可以取出表格中的常数。缺点是表格只能放在该条查表指令后面的256个单元之中，表格大小受到限制，而且表格只能被一段程序所利用。

第二条指令是以数据指针寄存器DPTR作为基址寄存器，累加器A的内容作为无符号数偏移量与DPTR的内容相加，得到一个16位的地址，并将该地址指出的程序存储器单元的内容送入累加器A。这条查表指令的执行结果只与DPTR和累加器A的内容有关，而与该条指令存放的地址及常数表格存放的地址无关，因此表格的大小和位置可以在64KB的程序存储器中任意安排，并且一个表格可以为各个程序块所公用。

7. 字节交换指令

```
XCH A, Rn          ;n=0~7
XCH A, direct
XCH A, @Ri         ;i=0或1
```

这组指令的功能是将累加器A的内容和源操作数的内容相互交换。

8. 半字节交换指令

```
XCHD A, @Ri        ;i=0或1
```

这条指令的功能是将累加器A的低4位内容和R(i)所指出的内部RAM单元的低4位内容相互交换。

1.8.4 控制转移指令

1. 无条件短跳转指令

```
AJMP addr11
```

这是2K字节范围内的无条件跳转指令，把程序存储器划分为32个区，每个区为2K字节，转移的目标地址必须与AJMP后面一条指令的第一个字节在同一个2K字节的范围之内（即转移目标地址必须与AJMP下一条指令的地址A15~A11相同），否则将引起混乱。该指令执行时，先将PC的内容加2，然后将11位地址送入PC.10~PC.0，而PC.15~PC.11保持不变。

2. 相对转移指令

```
SJMP rel
```

这是一条无条件跳转指令，执行时在(PC)+2后，把指令中有符号偏移量rel加到PC上，计算出偏移地址。因此转移的目标地址可以在这条指令前128个字节到后127个字节之间。

3. 长跳转指令

```
LJMP  addr16
```

这条指令执行时把指令的第2字节和第3字节分别装入PC的高8位和低8位字节中，无条件地转向指定的地址。转移的目标地址可以在64kB程序存储器地址空间的任何地方。

4. 散转指令

```
JMP  @A+DPTR
```

这条指令的功能是把累加器A中的8位无符号数与数据指针DPTR中的16位数相加，结果作为下一条指令的地址送入PC，不改变累加器A和数据指针DPTR的内容，也不影响标志。

条件转移指令是当满足某一特定条件时执行转移操作的指令。条件满足时转移（相当于一相对转移指令），条件不满足时则顺序执行下面一条指令。转移的目的地址在以下一条指令的起始地址为中心的256个字节范围之内（-128~+127）。当条件满足时，把PC的值加到下一条指令的第一个字节地址，再把有符号的相对偏移量rel加到PC上，计算出转移地址。

5. 条件转移指令

JZ	rel	; (A)=0时转移
JNZ	rel	; (A)≠0时转移
JC	rel	; CY=1时转移
JNC	rel	; CY=0时转移
JB	bit, rel	; (bit)=1时转移
JNB	bit, rel	; (bit)=0时转移
JBC	bit, rel	; (bit)=1时转移，并清"0"bit位

8051单片机没有专门的比较指令，但是提供了如下四条比较不相等转移指令。

6. 比较不相等转移指令

```
CJNE A, direct, rel
CJNE A, #data, rel
CJNE Rn, #data, rel ; n=0~7
CJNE @Ri, #data, rel ; i=0或1
```

这组指令的功能是比较前面两个操作数的大小。如果它们的值不相等，则转移。在把PC的值加到下一条指令的起始地址后，再把指令最后一个字节的有符号相对偏移量加到PC上，计算出转移地址。如果第一操作数（无符号整数）小于第二操作数（无符号整数），则置“1”进位标志CY，否则清“0”CY。不影响任何一个操作数的内容。

7. 减1不为0转移指令

```
DJNZ Rn, rel ; n=0~7
DJNZ direct, rel
```

这组指令把源操作数（Rn、direct）的内容减1，并将结果回送到源操作数中去。如果相减的结果不为0，则转移到由相对偏移量rel计算得到的目的地址。

8051单片机提供了两条子程序调用指令，即短调用和长调用指令。

8. 短调用指令

```
ACALL addr11
```

这是一条2K字节范围内的子程序调用指令。执行时，先把PC的值加2，获得下一条指令的地址，然后把获得的16位地址压进堆栈（PCL先进栈，PCH后进栈），并将堆栈指针SP的值加2，最后把PC值的高5位与指令提供的11位地址addr11相连接（PC15~PC11，a10~a0），形成子程序的入口地址并送入PC，使程序转向执行子程序。所调用的子程序的起始地址必须与ACALL指令后面一条指令的第一个字节在同一个2KB区域的程序存储器中。

9. 长调用指令

```
LCALL addr16
```

这条指令无条件地调用位于16位地址addr16处的子程序。它把PC的值加3以获得下条指令的地址并将其压入堆栈（先低位字节后高位字节），同时把SP的值加2，接着把指令的第2字节和第3字节（A15~A8，A7~A0）分别装入PC的高8位和低8位字节中，然后从PC所指出的地址开始执行程序。LCALL指令可以调用64K字节范围内程序存储器中的任何一个子程序，不影响任何标志。

10. 子程序返回指令

```
RET
```

这条指令的功能是从堆栈中弹出PC的高8位和低8位字节，同时把SP的值减2，并从PC指向的地址开始继续执行程序，不影响任何标志。

11. 中断返回指令

```
RETI
```

这条指令的功能与RET指令相似，不同的是它还清“0”单片机的内部中断状态标志。

12. 空操作指令

```
NOP
```

这条指令只完成(PC)+1操作，而不执行任何其他操作。

1.8.5 位操作指令

8051单片机内部RAM中有一个位寻址区，还有一些特殊功能寄存器也可以位地址，为此提供了丰富的位操作指令。

1. 位数据传送指令

```
MOV C, bit
```

```
MOV bit, C
```

这组指令的功能是把由源操作数指出的位变量送到目的操作数指定的位单元去。其中，一个操作数必须为进位标志，另一个操作数可以是任何可寻址位。

2. 位变量修改指令

```
CLR C      ;0→CY
CLR bit    ;0→bit
CPL C      ;对CY的内容取反
CPL bit    ;对bit位取反
SETB C     ; "1"→CY
SETB bit   ; "1"→bit
```

这组指令对操作数所指出的位进行清“0”、取反、置“1”的操作，不影响其他标志。

3. 位变量逻辑与指令

```
ANL C, bit
ANL C,  $\overline{\text{bit}}$ 
```

这组指令的功能是将进位标志与指定的位变量（或位变量的取反值）相“与”，结果送到进位标志，不影响别的标志。

4. 位变量逻辑或指令

```
ORL C, bit
ORL C,  $\overline{\text{bit}}$ 
```

这组指令的功能是将进位标志与指定的位变量（或位变量的取反值）相“或”，结果送到进位标志，不影响别的标志。

附录A按指令功能列出了8051的全部指令。

1.9 汇编语言程序设计

前面介绍了8051单片机的指令系统，在实际应用中将这些指令按需要有序地排列成一段完整的程序，就可以完成某一特定的任务。通常把这种程序称为汇编语言源程序。它主要由指令助记符和一些汇编伪指令组成，而把可以直接在计算机上运行的机器语言程序称为目标代码，由汇编语言源程序转换为目标代码的过程称为“汇编”，可以通过查附录A的指令表将汇编语言源程序中的指令逐条翻译为机器代码。实际上，现在已经有许多在个人计算机上运行的专门“汇编程序”（如ASM51等），可以很方便地将汇编语言源程序转换成目标代码。

8051单片机汇编语言程序由若干条指令行组成，一般格式为

```
[标号:] 操作码, [操作数] [; 注释]
```

其中各段意义说明如下：

- “标号”是可选项，可用来表示程序的地址；

- “操作码”是8051单片机的指令助记符。
- “操作数”是可选项，依赖于不同的8051指令，有些指令不需要操作数，有些指令需要1~3个操作数，操作数可以是数字、符号或地址。十进制数以字符“D”为后缀，十六进制数以字符“H”为后缀，八进制数以字符“O”为后缀，二进制数以字符“B”为后缀，省略后缀时则默认为十进制数。立即数的前面须冠以符号“#”。
- “注释”也是可选项，是为理解程序含义而加上的文字解释，注释文字前面必须有一个分号。

在对汇编语言源程序进行汇编时，8051指令行将被转换为一一对应的目标代码，它们可以被单片机CPU执行。另外，汇编语言源程序中还包含一些不能被单片机CPU执行的指令，称为汇编伪指令。它们仅提供汇编控制信息，用于在汇编过程中执行一些特殊操作，而不会被转换为目标代码。下面介绍一些常用的汇编伪指令。

1. 设置起始地址 ORG

一般格式：ORG nnnn

其中，nnnn为4位十六进制数，表示程序的起始地址。ORG伪指令总是出现在每段程序的开始处，用于对该段程序在程序存储器中进行定位。需要注意的是，由ORG设置的程序空间地址应从小到大，并且不能重复，如

```
ORG 1000H
MAIN: MOV A, 20H
...
```

表示该段程序在程序存储器中的起始地址为1000H。换句话说，这里标号“MAIN”所代表的就是地址值1000H。

2. 定义字节DB

一般格式：[标号:] DB 项或项表

其中，“项或项表”是单个字节数据，或多个由逗号隔开的单字节数据，它们可以是数值，也可以是用引号括起来的ASCII字符串。DB伪指令的功能是将项或项表的数据存入由标号（地址）开始的连续存储器单元之中，如

```
ORG 1000H
SEG1: DB 53H, 78H
SEG2: DB 'THIS IS A TEST'
```

注意，项或项表若为数值，则其范围为00H~FFH，若为字符串，则其长度不能超过80个字符。

3. 定义字DW

一般格式：[标号:] DW 项或项表

DW的基本含义与DB相似，不同之处在于DW用于定义16位数据，如

```
ORG 1000H
```

TABLE: DW 1234H, 78H

4. 保留存储器空间DS

一般格式: [标号:] DS 表达式

DS伪指令的功能是从标号指定的存储器地址开始, 保留由表达式的值规定的存储器空间单元, 如

```
ORG 1000H
TEMP: DS 10
```

本例表示从TEMP地址(1000H)开始保留10个连续的存储器单元。

5. 为标号赋值EQU

一般格式: 字符名 EQU 表达式

EQU伪指令的功能是将表达式的值赋给“字符名”。“字符名”一旦被赋值之后, 它的值在整个程序中就不能再改变。注意, 这里“字符名”与标号不同, 它后面没有冒号, 如

```
PPAGE EQU 9000H
EN EQU 1
```

6. 源程序结束END

一般格式: END

END是一个程序结束标志, 通常放在汇编语言源程序的结尾。

汇编语言程序设计, 也就是编写汇编语言源程序。源程序文件的扩展名为.asm或.a51。汇编语言程序有主程序和子程序之分。主程序又称为前台程序, 通常是一个无穷循环, 完成单片机系统的初始化, 如内存单元清零、功能设置、开放中断等。子程序又称为后台程序, 可以是各种功能子程序, 也可以是中断服务子程序, 通常被其他程序所调用, 完成某个具体任务, 如数据采集存储、科学计算、时间延时等。一般在前台主程序的循环体中根据需要不断调用各种后台功能子程序, 从而完成单片机应用系统规定的任务。

下面是一个汇编语言设计的例子。

例1-1 用汇编语言程序实现 $c=a^2+b^2$ 。假设a、b、c分别存放于单片机片内RAM的30H、31H、32H三个单元。主程序通过调用子程序“SQR”用查表方式分别求得 a^2 和 b^2 的值, 然后进行相加得到最后的c值。

主程序为

```
ORG 0000H          ; 程序的复位入口
START: LJMP MAIN
ORG 0030H          ; 主程序入口
MAIN: MOV 30H, #03   ; a=3
      MOV 31H, #04   ; b=4
      MOV A, 30H     ; 取得a值
      LCALL SQR      ; 调查表子程序
```

```
MOV R1,A           ; a2暂存于R1中
MOV A,31H          ; 取得b值
LCALL SQR          ; 调查表子程序
ADD A,R1           ; 计算a2+b2
MOV 32H,A          ; 存结果
WAIT: SJMP $        ; 循环, 等待
```

查表子程序为

```
ORG 0F00H
SQR: MOV DPTR,#TAB
      MOVC A,@A+DPTR ; 查表求得平方值
      RET            ; 子程序返回
TAB: DB 0,1,4,9,16   ; 平方表
      DB 25,36,49,64,81
      END            ; 程序结束
```


Proteus虚拟仿真

英国Labcenter公司推出的Proteus软件采用虚拟仿真技术，很好地解决了单片机及其外围电路的设计和协同仿真问题，可以在没有单片机实际硬件的条件下，利用个人计算机实现单片机软件和硬件同步仿真，仿真结果可以直接应用于真实设计，极大地提高了单片机应用系统的设计效率，同时也使得单片机的学习和应用开发过程变得容易和简单。Proteus软件包提供了丰富的元器件库，可以根据不同要求设计各种单片机应用系统。Proteus软件已有近20年的历史，它针对单片机的应用，可以直接在基于原理图的虚拟模型上进行软件编程和虚拟仿真调试，配合虚拟示波器、逻辑分析仪等，用户能看到单片机系统运行后的输入、输出效果。Proteus在国外已经得到广泛使用，国内一些高校和公司也开始尝试使用该软件进行单片机教学和系统设计，在不需要专门硬件投入的前提下，利用个人计算机来学习单片机知识，比单纯从书本学习更易于接受和提高，还可以增加实际编程经验。

2.1 集成环境ISIS

Proteus软件包提供一种界面友好的人机交互式集成环境ISIS。其设计功能强大，使用方便。ISIS在Windows环境下运行，启动后弹出如图2.1所示界面，由下拉菜单、快捷工具栏、预览窗口、原理图编辑窗口、元器件列表窗口、元器件方向选择、仿真按钮组成。

下拉菜单提供如下功能选项：

File菜单包括常用的文件功能，如创建一个新设计、打开已有设计、保存设计、导入/导出文件、打印设计文档等。

View菜单包括是否显示网格、设置网格间距、缩放原理图、显示与隐藏各种工具栏等。

Edit菜单包括撤销/恢复操作、查找与编辑、剪切、复制、粘贴元器件、设置多个对象的层叠关系等。

Library菜单包括添加、创建元器件/图标、调用库管理器。

Tools菜单包括实时标注、实时捕捉、自动布线等。

Design菜单包括编辑设计属性、编辑图纸属性、进行设计注释等。

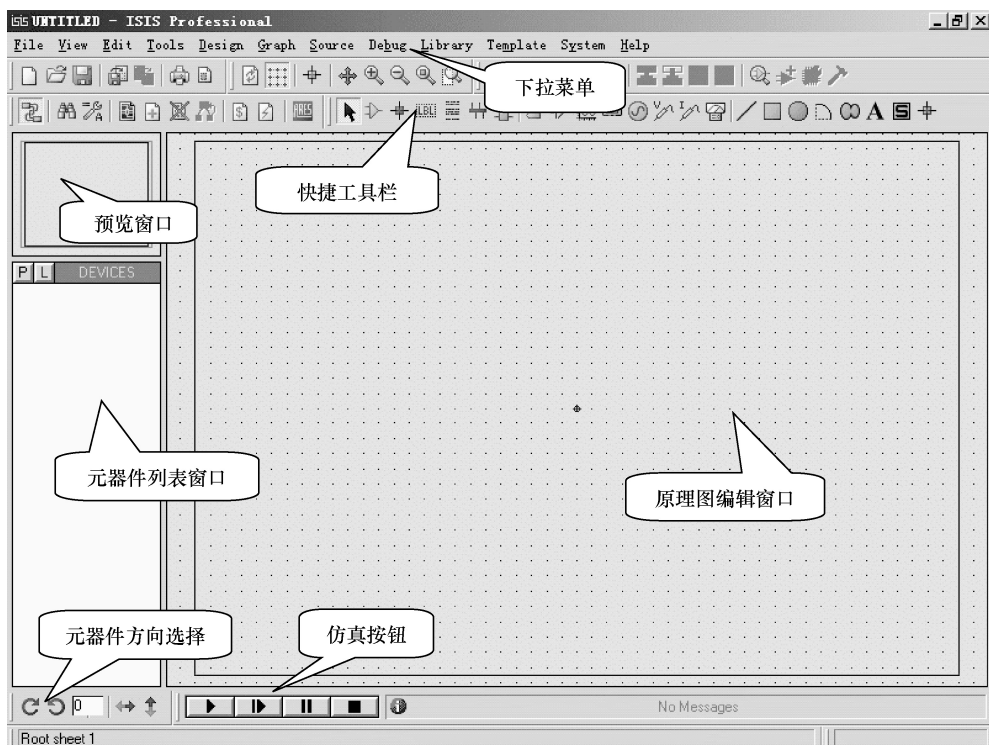


图2.1 ISIS环境界面

Graph菜单包括编辑图形、添加Trace、仿真图形、一致性分析等。

Source菜单包括添加/删除源程序文件、定义代码生成工具、调用外部文本编辑器等。

Debug菜单包括启动调试、进行仿真、单步执行、重新排布弹出窗口等。

Template菜单包括设置图形格式、文本格式、设计颜色、节点形状等。

System菜单包括设置环境变量、工作路径、图纸尺寸大小、字体、快捷键等。

Help菜单包括版权信息、帮助文件、例程等。

快捷工具栏分为主工具栏和元器件工具栏。主工具栏包括文件工具、视图工具、编辑工具、设计工具4个部分。每个工具栏提供若干快捷按钮。

主工具按钮如图2.2所示。从左往右各按钮功能依次为：

- 新建设计；
- 打开已有设计；
- 保存设计；
- 导入文件；
- 导出文件；
- 打印设计文档；
- 标识输出区域。



图2.2 主工具按钮

视图工具按钮如图2.3所示。从左往右各按钮功能依次为：

- 刷新；
- 网格开关；
- 原点；
- 选择显示中心；
- 放大；
- 缩小；
- 全图显示；
- 区域缩放。



图2.3 视图工具按钮

编辑工具按钮如图2.4所示。从左往右各按钮功能依次为：

- 撤销；
- 重做；
- 剪切；
- 复制；
- 粘贴；
- 复制选中对象；
- 移动选中对象；
- 旋转选中对象；
- 删除选中对象；
- 从器件库选元器件；
- 制作器件；
- 封装工具；
- 释放元件。



图2.4 编辑工具按钮

设计工具按钮如图2.5所示。从左往右各按钮功能依次为：

- 自动布线；
- 查找；
- 属性分配工具；
- 设计浏览器；
- 新建图纸；
- 删除图纸；
- 退到上层图纸；
- 生成元件列表；

- 生成电气规则检查报告；
- 创建网络表。



图2.5 设计工具按钮

元器件工具栏包括方式选择、配件模型、绘制图形3个部分，每个工具栏提供若干快捷按钮。

模型选择按钮如图2.6所示。从左往右各按钮功能依次为：

- 选择即时编辑元件；
- 选择放置元件；
- 放置节点；
- 放置网络标号；
- 放置文本；
- 绘制总线；
- 放置子电路图。



图2.6 模型选择按钮

配件模型按钮如图2.7所示，从左往右各按钮功能依次为：

- 端点方式，有 VCC、地、输出、输入等；
- 器件引脚方式，用于绘制各种引脚；
- 仿真图表；
- 录音机；
- 信号发生器；
- 电压探针；
- 电流探针；
- 虚拟仪表。



图2.7 配件模型按钮

图形模型按钮如图2.8所示。从左往右各按钮功能依次为：

- 绘制直线；
- 绘制方框；
- 绘制圆；
- 绘制圆弧；
- 绘制多边形；
- 编辑文本；

- 绘制符号；
- 绘制原点。



图2.8 图形模型按钮

在元器件列表窗口下方有一个元器件方向选择按钮栏，按钮如图2.9所示。从左往右各按钮功能依次为：

- 向右旋转 90 度；
- 向左旋转 90 度；
- 水平翻转；
- 垂直翻转。



图2.9 元器件方向选择按钮

在原理图编辑窗口下方有一个仿真工具按钮栏，按钮如图2.10所示。从左往右各按钮功能依次为：

- 全速运行；
- 单步运行；
- 暂停；
- 停止。



图2.10 仿真工具按钮

原理图编辑窗口是用来绘制原理图的，蓝色方框内为编辑区，里面可以放置元器件和连线。注意，这个窗口没有滚动条，需要用预览窗口来改变原理图的可视范围，也可以用鼠标滚轮对显示内容进行缩放。

预览窗口可显示两种内容：一种是在元器件列表窗口选中某个元器件时，将显示该元器件的预览图；另一种是当鼠标落在原理图编辑窗口时（即放置元器件到原理图编辑窗口后或在原理图编辑窗口中单击鼠标后），将显示整张原理图的缩略图，并显示一个绿色的方框，绿色方框里面就是当前原理图编辑窗口中显示的内容，可用鼠标改变绿色方框的位置，从而改变原理图的可视范围。

2.2 绘制原理图

绘制原理图是在原理图编辑窗口中蓝色方框内完成的，通过下拉菜单System中Set Sheet Size选项，可以调整原理图设计页面大小。绘制原理图时，首先应根据需要选取元器件，Proteus ISIS库中提供了大量元器件原理图符号，利用Proteus ISIS的搜索功能能很方便地查找需要的元器件。下面以如图2.11所示为例来说明绘制原理图的方法。

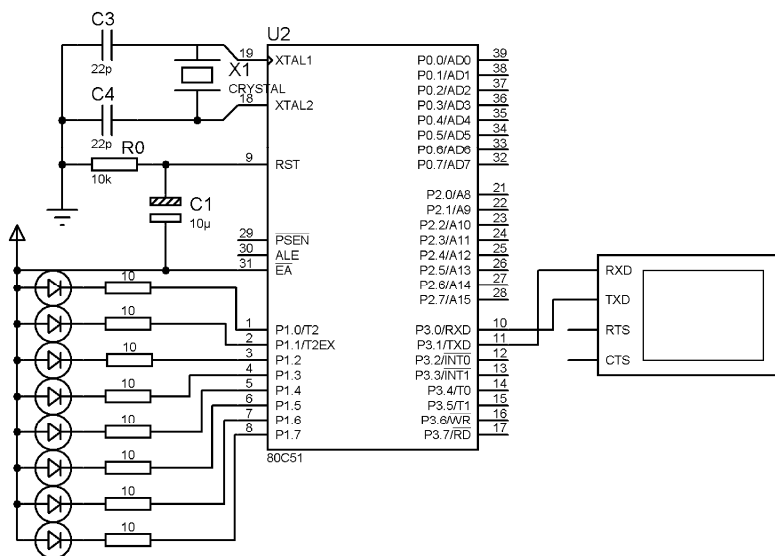


图2.11 绘制原理图示例

首先根据需要选择元器件。单击元器件列表窗口上边的按钮P，弹出如图2.12所示元器件选择窗口，在该窗口左上方的“Keywords”栏内键入8051，窗口中间的“Results”栏将显示出元器件库中所有8051单片机，选择其中的80C51，窗口右上方的“80C51 Preview”栏将显示出80C51图形符号，同时显示该器件的虚拟仿真模型“VSM DLL Model (MCS8051.DLL)”。单击“OK”后，选择的元器件将出现在器件列表窗口。照此办理，选择所有需要的元器件，如果选择的元器件显示“No Simulator Model”，则说明该器件没有仿真模型，将不能进行虚拟仿真。

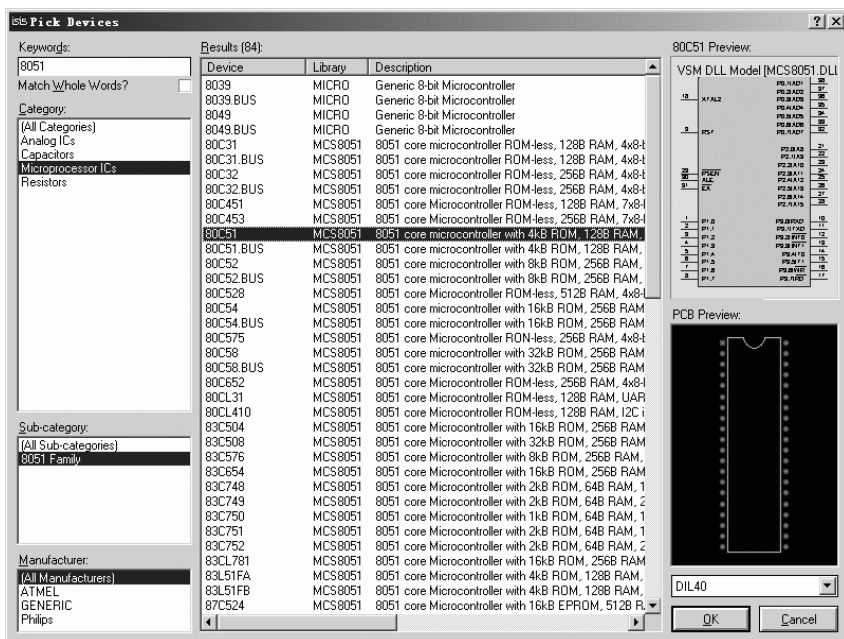


图2.12 元器件选择窗口

如果遇到库中没有的元器件，就需要自己创建。通常有两种方法创建自己的元器件：一种是用PROTEUS VSM SDK开发仿真模型，并制作元器件；另一种是在已有的元器件基础上进行改造。关于具体的创建方法这里不做介绍，请读者查阅相关资料。

元器件选择完毕后，就可以开始绘制原理图了。先用鼠标从元器件选择窗口选中需要的元器件，预览窗口将出现该元器件的图标，再将鼠标指向编辑窗口并单击左键，将选中的元器件放置到原理图中。

放置电源和地线端时，要从配件模型按钮栏中选取。

在两个元器件之间进行连线的方式很简单，先将鼠标指向第一个元器件的连接点并单击左键，再将鼠标移到另一个元器件的连接点并单击左键，这两个点就被连接到了一起。对于相隔较远，直接连线不方便的元器件，可以用标号的方式连接。

在编辑窗口中绘制原理图的一般操作总结如下：用左键放置元器件，右键选择元器件，双击右键删除元器件，右键拖选多个元器件，先右键后左键编辑元器件属性，先右键后左键拖动元器件，连线用左键，删除用右键，中键缩放整个原理图。

2.3 创建汇编语言源代码仿真文件

Proteus虚拟仿真系统将汇编语言源代码的编辑与编译整合在同一设计环境中，使用户可以在设计中直接编辑汇编语言源程序和生成仿真代码，并且很容易察看源程序经过修改之后对仿真结果的影响。Proteus软件包自带多种汇编语言工具，对于生成汇编语言源程序仿真代码十分方便。使用时，先要设置代码生成工具，单击Source下拉菜单中“Define Code Generaation Tools”选项，弹出如图2.13所示定义代码生成工具窗口。

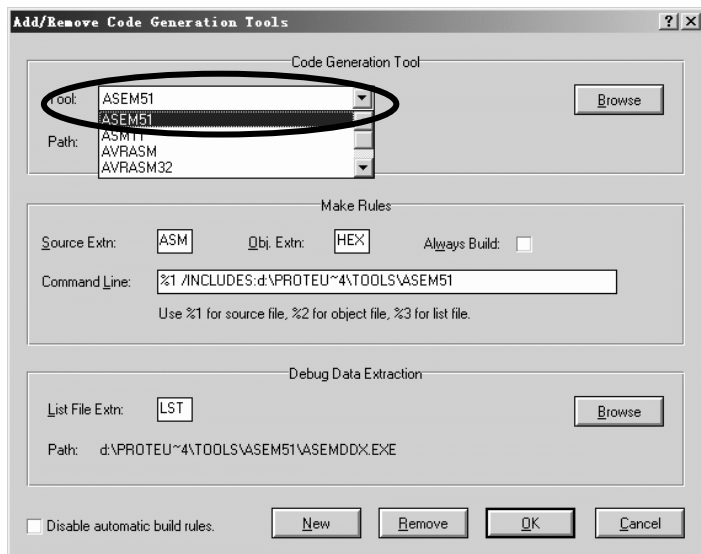


图2.13 定义代码生成工具窗口

在“Code Generation Tool”栏的Tool对话框内选择“ASEM51”，设定8051单片机汇编工具；在“Make Rules”栏的“Source Extn”对话框内选择ASM，在“Obj Extn”对话框内

选择HEX，设定源程序扩展名和目标代码扩展名；在“Debug Data Extraction”栏的“List File Extn”对话框内选择LST，设定列表文件扩展名，设置完成后，单击“OK”按钮退出。

采用文本编辑器编写例2-1所示汇编语言源程序，并保存为ex2-1.asm。

例2-1 汇编语言流水灯程序文件。

```

ORG 0000H
START:MOV R7,#08H
      MOV A,#0FEH ;正向流水灯
LP1:MOV P1,A
      LCALL DELAY
      RL A
      DJNZ R7,LP1
      MOV R7,#08H ;反向流水灯
      MOV A,#07FH
LP2:MOV P1,A
      LCALL DELAY
      RR A
      DJNZ R7,LP2
      LJMP START
DELAY:MOV R6,#0FFH
LP3:MOV R5,#0FFH
LP4:DJNZ R5,LP4
      DJNZ R6,LP3
      RET
      END

```

接着要添加源程序文件，单击Source下拉菜单中“Add/Remove Source File”选项，弹出如图2.14所示添加/删除源程序文件窗口，在“Code Generation Tool”栏内选择ASEM51，再单击“New”按钮，弹出如图2.15所示源程序文件查找窗口，在“查找范围”栏选中源程序文件的保存文件夹，同时在“文件名”栏键入源程序名，如果该源程序文件已经存在，则单击“打开”按钮即完成源程序文件的添加，如果该源程序文件不存在，则单击“打开”按钮后将弹出如图2.16所示提示对话框，询问是否创建该文件，单击“是”按钮即在选择的文件夹内创建一个新文件。文件添加完成后，再单击Source下拉菜单，可以看到源程序文件已经位于其中，如图2.17所示，此时可以直接单击文件名将其打开进行编辑或修改。

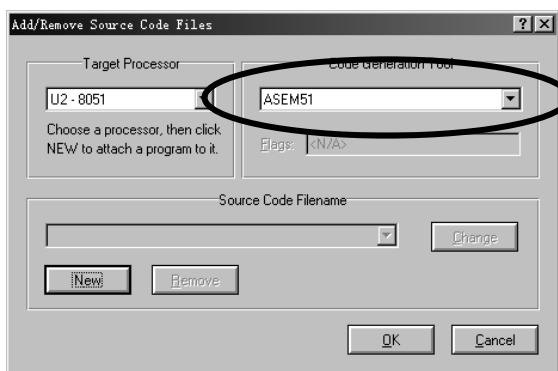


图2.14 添加/删除源程序文件窗口

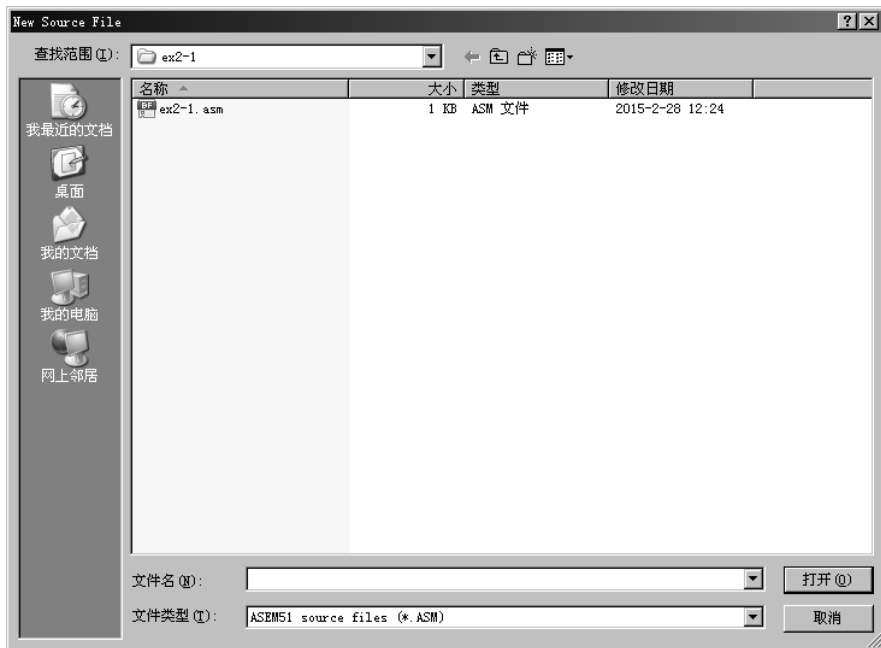


图2.15 源程序文件查找窗口

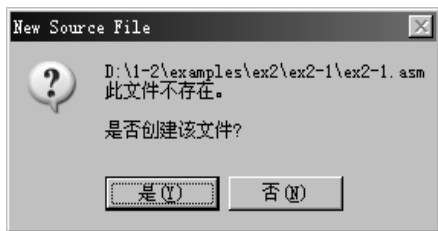


图2.16 新建源程序对话框

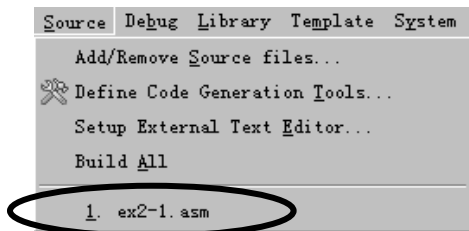




图2.17 源程序文件被添加到Source菜单

已添加的源程序文件编辑修改完成后，单击Source菜单中的“Build All”选项对文件进行汇编连接，生成可执行的十六进制文件（.Hex）、列表文件（.LST）和源代码仿真调试文件（.SDI）。

2.4 在原理图中进行源代码仿真调试

原理图绘制完成后，给单片机添加应用程序，就可以进行虚拟仿真调试了。先用鼠标右键选中8051单片机，再单击左键，弹出如图2.18所示器件编辑窗口。在器件编辑窗口中“Program File”栏单击文件夹浏览按钮，找到需要仿真的Hex文件，单击确定按钮完成添加文件，在“Clock Frequency”中把频率改为12MHz，单击“OK”退出。这时单击仿真工具栏中全速运行按钮即可开始进行虚拟仿真。为了直观地看到仿真结果，还可以在原理图中添加一些虚拟仪表，可用的虚拟仪表有电压表、电流表、虚拟示波器、逻辑分析仪、计数器/定时器、虚拟终端、虚拟信号发生器、序列发生器、I²C调试器、SPI调试器等。

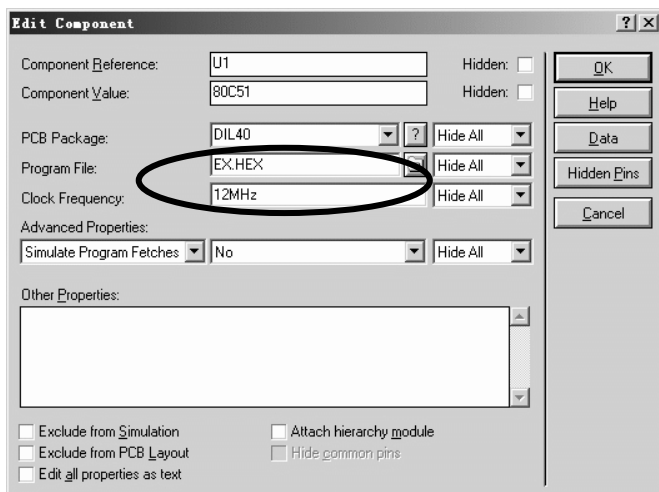









图2.18 器件编辑窗口

在全速运行过程中单击仿真工具中的暂停按钮，可暂停程序运行，再单击Debug下拉菜单中的“6. 8051 CPU Source Code”选项，弹出如图2.19所示源代码调试窗口。

源代码调试窗口右上角提供如下一些调试按钮。

-  全速运行 (Run)。启动程序全速运行。
-  单步运行 (Step Over)。执行子程序调用指令时，将整个子程序一次执行完。
-  跟踪运行 (Step Into)。遇到子程序调用指令时，跟踪进入子程序内部运行。
-  跳出运行 (Step Out)。将整个子程序运行完成，并返回到主程序。
-  运行到光标处 (Run To)。从当前指令运行到光标所在位置。
-  设置断点 (Toggle Breakpoint)。将光标所在位置设置一个断点。

将鼠标指向源代码调试窗口并单击右键，将弹出如图2.20所示右键快捷菜单，提供如下功能选项。

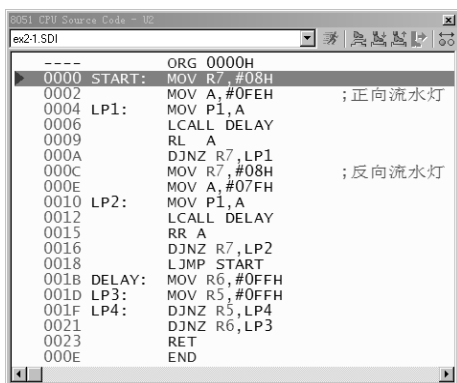


图2.19 源代码调试窗口

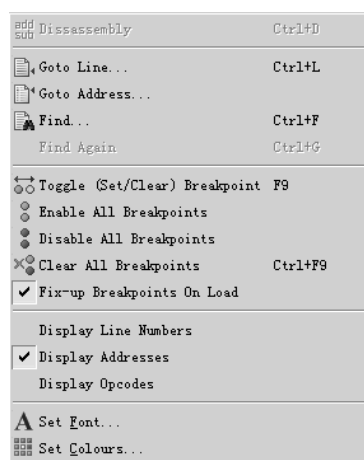


图2.20 源代码调试窗口中的右键菜单

“Goto Line...”，单击该选项，在弹出的对话框中键入源程序代码的行号，光标立即跳转到指定行。

“Goto Address...”，单击该选项，在弹出的对话框中键入源程序代码的地址，光标立即跳转到指定地址处。

“Find...”，单击该选项，在弹出的对话框中键入希望查找的文本字符，将在源代码调试窗口从当前光标所在位置开始查找指定的字符。

“Find Again...”，将重复上次查找内容。

“Toggle Breakpoint”，在光标所在处设置或删除断点。

“Enable All Breakpoints”，允许所有断点。

“Disable All Breakpoints”，禁止所有断点。

“Clear All Breakpoints”，清除所有断点。

“Fix-up All Breakpoints On Load”，装入时修复断点。

“Display Line Numbers”，显示行号。

“Display Addresses”，显示地址。

“Display Opcode”，显示操作码。

“Set Font...”，单击该选项，在弹出的对话框中设置源代码调试窗口中显示字符的字体。

“Set Colour...”，单击该选项，可设置弹出窗口的颜色。

在进行源代码调试时，Debug下拉菜单提供了多种弹出窗口，给调试过程带来了许多方便。单击Debug下拉菜单中的“5. 8051 CPU Internal (IDATA) Memory”选项，弹出如图2.21所示8051单片机片内存储器窗口，显示当前片内存储器的内容。

单击Debug下拉菜单中的“4. 8051 CPU SFR Memory”选项，弹出如图2.22所示8051单片机特殊功能寄存器窗口，显示当前特殊功能寄存器的内容。

Address	Hex	ASCII
00	00 00 00 00
08	00 00 00 00
10	00 00 00 00
18	00 00 00 00
20	00 00 00 00
28	00 00 00 00
30	00 00 00 00
38	00 00 00 00
40	00 00 00 00
48	00 00 00 00
50	00 00 00 00
58	00 00 00 00
60	00 00 00 00
68	00 00 00 00
70	00 00 00 00
78	00 00 00 00

图2.21 片内存储器窗口

Address	Hex	ASCII
80	FF 07 00 00
88	00 00 00 00
90	FF 00 00 00
98	00 00 00 00
A0	FF 00 00 00
A8	00 00 00 00
B0	FF 00 00 00
B8	00 00 00 00
C0	00 00 00 00
C8	00 00 00 00
D0	00 00 00 00
D8	00 00 00 00
E0	00 00 00 00
E8	00 00 00 00
F0	00 00 00 00
F8	00 00 00 00

图2.22 特殊功能寄存器窗口

单击Debug下拉菜单中的“3. 8051 CPU Register”选项，弹出如图2.23所示8051单片机寄存器窗口，显示当前各个寄存器的值。

上述各个窗口的内容随着调试过程自动发生变化，在单步运行时，发生改变的值会高亮显示，显示格式可以通过相应的窗口提供右键菜单选项进行调整。在全速运行时，上述各窗口将自动隐藏。

单击Debug下拉菜单中的“2. Watch Window”选项，弹出如图2.24所示观测窗口。观测窗口即使在全速运行期间也将保持实时显示，可以在观测窗口中添加一些项目，以便于程序调试期间进行察看。添加项目可以通过观测窗口中的右键菜单实现，也可以先在图2.21或图2.22中用鼠标左键标记希望进行观测的存储器单元，然后将其直接拖到观测窗口中。

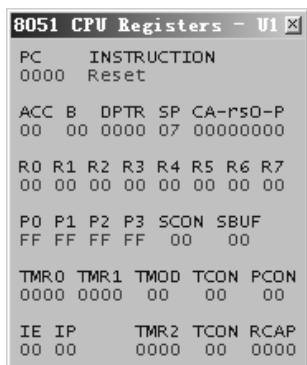


图2.23 寄存器窗口

Name	Address	Value	Watch ...
P0	0x0000	0xFF	
P1	0x0010	0x7F	
— 0x0000	0x0000	0x00	
— 0x0001	0x0001	0x00	
— 0x0002	0x0002	0x01	
— 0x0003	0x0003	0x00	
— 0x0004	0x0004	0x00	
— 0x...	0x0005	0x0E	
— 0x0006	0x0006	0x04	
— 0x...	0x0007	0x4E	
— 0x0008	0x0008	0x0A	

图2.24 观测窗口

2.5 原理图与Keil环境联机仿真调试

德国Keil Software公司多年来致力于单片机C语言编译器的研究。该公司开发的Keil C51是一种专为8051单片机设计的高效率C语言编译器，符合ANSI标准，生成的程序代码运行速度极高，所需要的存储器空间极小，完全可以与汇编语言相媲美。目前，Keil公司推出的C51编译器已被完全集成到一个功能强大的全新集成开发环境 μ Vision3中，包括项目管理、程序编译、连接定位等，并且还可以通过专门驱动软件（Proteus VSM Keil Debugger Driver）与Proteus原理图进行联机仿真，为单片机的开发带来极大的方便。驱动软件可以到labcenter网站免费下载。下面通过一个简单的实例说明采用Keil环境编写单片机C51程序及与Proteus原理图进行联机仿真调试的步骤。

启动 μ Vision3后，用鼠标左键单击“Project菜单/New Project”选项，在弹出的对话框窗口中输入项目文件名max，选择合适的保存路径并单击“保存”按钮，创建一个文件名为max.uv2的新项目文件，如图2.25所示。

图2.25 在 μ Vision3中新建一个项目

项目名保存完毕后，将弹出如图2.26所示器件数据库对话框窗口，根据需要选择CPU器件Atmel公司的AT89C51。

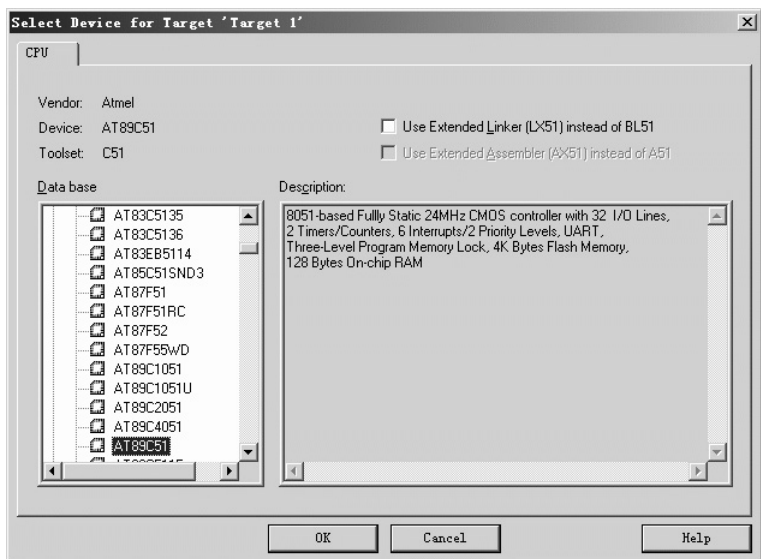


图2.26 为项目选择CPU器件

创建新项目后，会自动包含一个默认的目标Target 1和文件组Source Group 1，同时还会提示是否加入启动代码“Startup.a51”文件，一般应选择“是”。

接下来创建用户自己的应用程序，单击“File菜单/New”选项，从打开的编辑窗口中输入下面例2-2的C51源程序。

例2-2 求两个输入数据中较大者的C51源程序。

```
#include <reg51.h>           // 预处理命令
#include <stdio.h>
#define uint unsigned int
uint max (uint x, uint y);    // 功能函数max说明
main() {                      // 主函数
    uint a, A, c;             // 主函数的内部变量类型说明
    SCON=0x52; TMOD=0x20;     // 串行口、定时器初始化
    PCON=0x80; TH1=0x0F3;     // fosc=12MHz, 波特率=4800
    TL1=0x0F3; TCON=0x69;
    printf ( "\n Please enter two numbers: \n\n"); // 输出提示符
    scanf ("%d %d", &a, &A); // 输入变量a和b的值
    c= max (a,A);              // 调用max函数
    printf ( " \n max =%u \n ", c); // 输出较大数据的值
    while(1);
}                               // 主程序结束

uint max (uint x , uint y) {    // 定义max函数, x、y为形式参数
    if ( x > y ) return (x);    // 将计算得到的最大值返回到调用处
    else return(y);
}                               // max函数结束
```

程序输入完成后，保存为max.c的源程序文件，保存路径一般与项目文件相同。然后将

鼠标指向“项目窗口/Files”标签页中的“Source Group 1”文件组并单击右键，弹出右键菜单，如图2.27所示。

单击右键菜单中的“Add Files to Group ‘Source Group 1’”选项，将刚才保存的源程序文件max.c添加到项目中去，如图2.28所示。

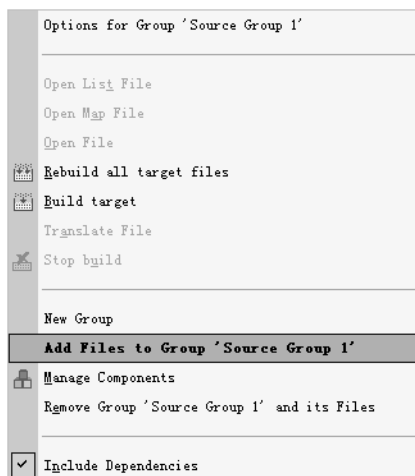


图2.27 项目窗口的右键菜单

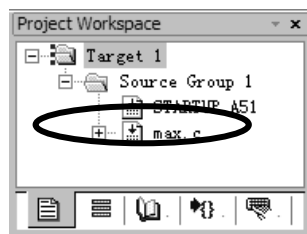


图2.28 添加程序文件到项目窗口

接下来需要对项目进行必要的配置。单击“Project菜单/Options for Target”选项，弹出如图2.29所示窗口。这是一个十分重要的窗口，包括“Device”、“Target”、“Output”、“Listing”、“C51”、“A51”、“BL51 Locate”、“BL51 Misc”和“Debug”等多个选项卡。其中一些选项可以直接用其默认值，也可进行适当调整。图2.29为其中的“Target”配置选项卡，用于设定目标硬件系统的时钟频率Xtal为12.0MHz、编译器的存储器模式为Small（C51程序中局部变量位于片内数据存储器RAM空间）、程序存储器ROM空间设为Large（使用64KB程序存储器）、不采用实时操作系统、不采用代码分组设计。

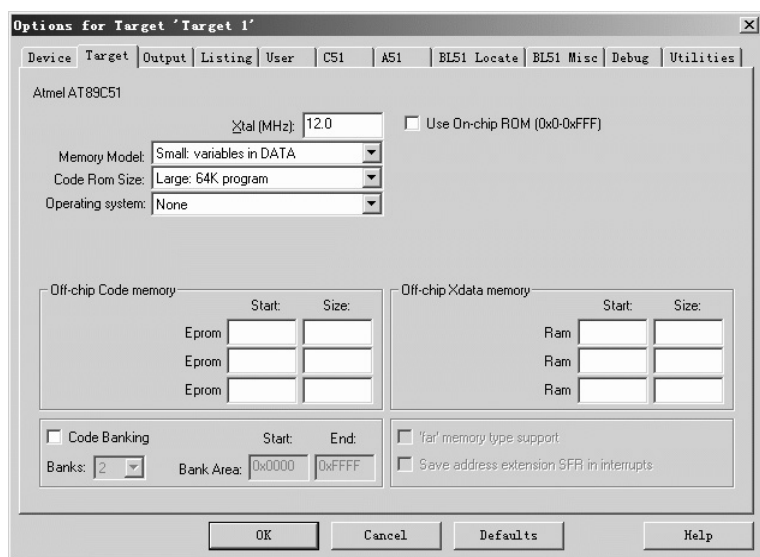


图2.29 Target配置选项卡

图2.30为“Output”配置选项卡，用于设定当前项目在编译连接之后生成的可执行代码文件，默认为与项目文件同名，也可以指定为其他文件名，存放在当前项目文件所在的目录中，也可以单击“Select Folder For Objects...”来指定文件的目录路径。选中方形复选框“Debug Information”将在输出文件中包含进行源程序调试的符号信息。选中方形复选框“Browse Information”将在输出文件中包含源程序浏览信息。选中方形复选框“Create HEX File”表示除了生成可执行代码文件之外，还将生成一个HEX文件。

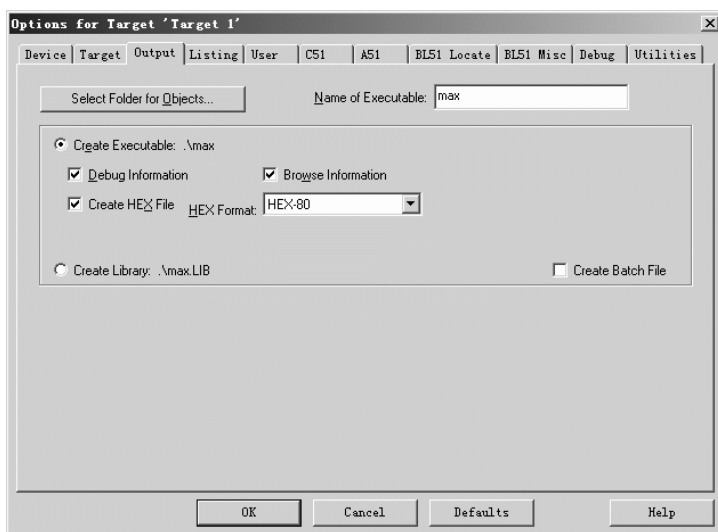


图2.30 Output配置选项卡

图2.31为“Debug”配置选项卡，用于设定 μ Vision3调试选项。 μ Vision3环境可以通过专用硬件驱动Proteus VSM Simulator与Proteus原理图进行联机仿真，为单片机开发带来极大的方便。Keil与Proteus联机之前，需要先安装Proteus VSM驱动（该驱动可以到Labcenter公司网站免费下载）。

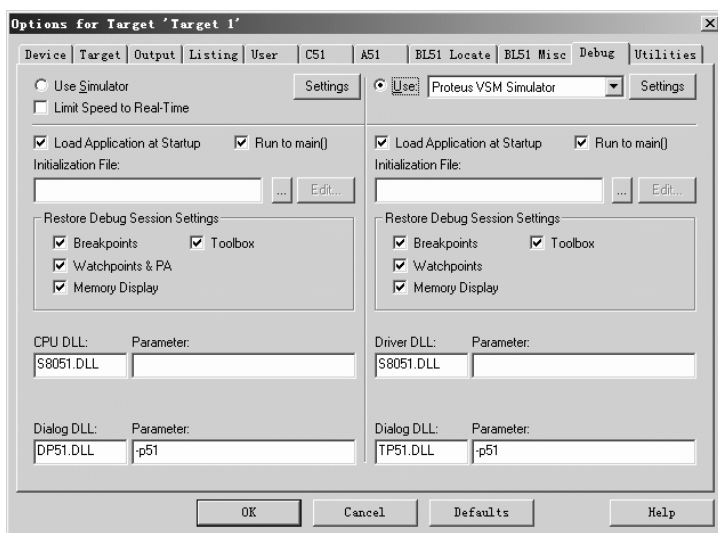


图2.31 Debug配置选项卡

单击选项卡右上部圆形复选框“Use”，通过文本框中的箭头，选择其中的“Proteus VSM Simulator”，再单击右边的“Settings”按钮，弹出如图2.32所示通信配置选项卡。需要注意的是，用户的Windows系统中必须安装TCP/IP 协议，才能保证Proteus与Keil正常通信。

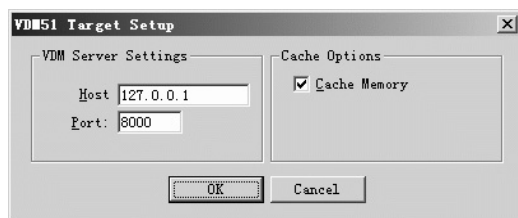


图2.32 Debug通信配置选项卡

单击如图2.31所示选项卡中“Load Application at Startup”和“Run to main()”复选框，可以在启动仿真时自动装入应用程序目标代码并运行到main()函数处。在“Restore Debug Session Settings”栏中有4个方形复选框：“Breakpoints”、“Watchpoints”、“Memory Display”和“Toolbox”，分别用于在启动Debug调试器时自动恢复上次调试过程中所设置的断点、观察点与性能分析器、存储器及工具箱的显示状态，如果在编辑源程序文件时就设置了断点并希望在启动Debug仿真调试时能够使用，则应该选中这些复选框。

完成上述基本选项配置之后，将鼠标指向项目窗口中的文件“max.c”并单击右键，从弹出的右键菜单中单击“Build target”选项， μ Vision3将按以上选项配置自动完成对当前项目中的编译链接，并在输出窗口中显示提示信息，如图2.33所示。

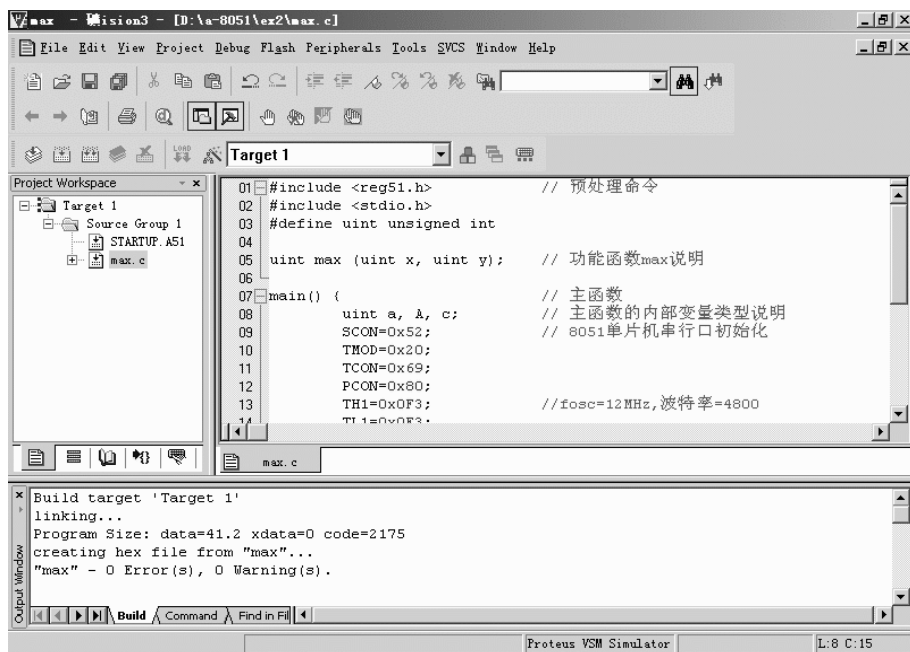


图2.33 编译链接完成后输出窗口的提示信息

C51程序编译链接完成后，先打开Proteus原理图，单击ISIS环境的“Debug菜单/Use Remote Debug Monitor”选项，准备与Proteus原理图进行联机仿真调试，如图2.34所示。

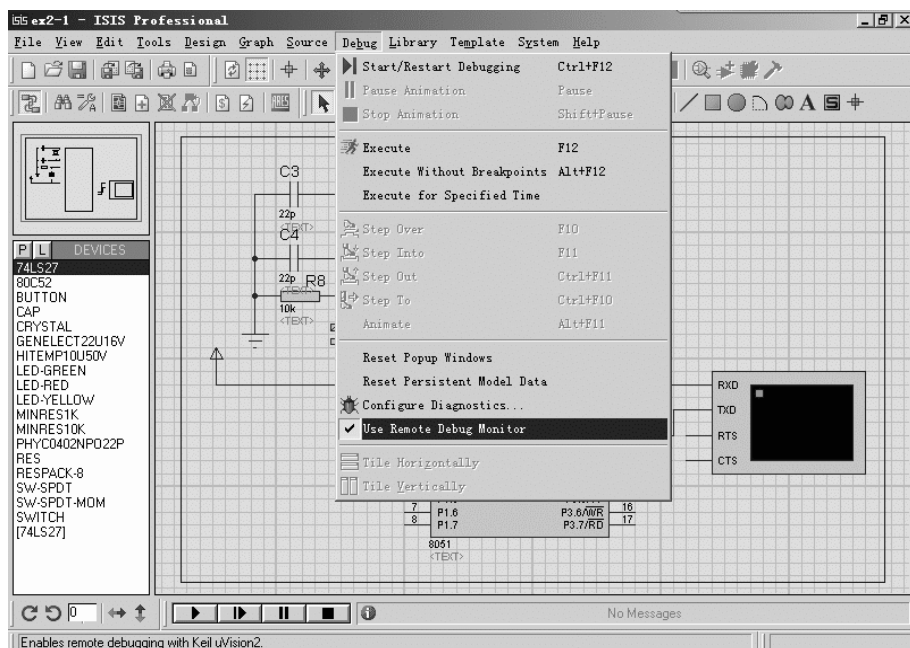
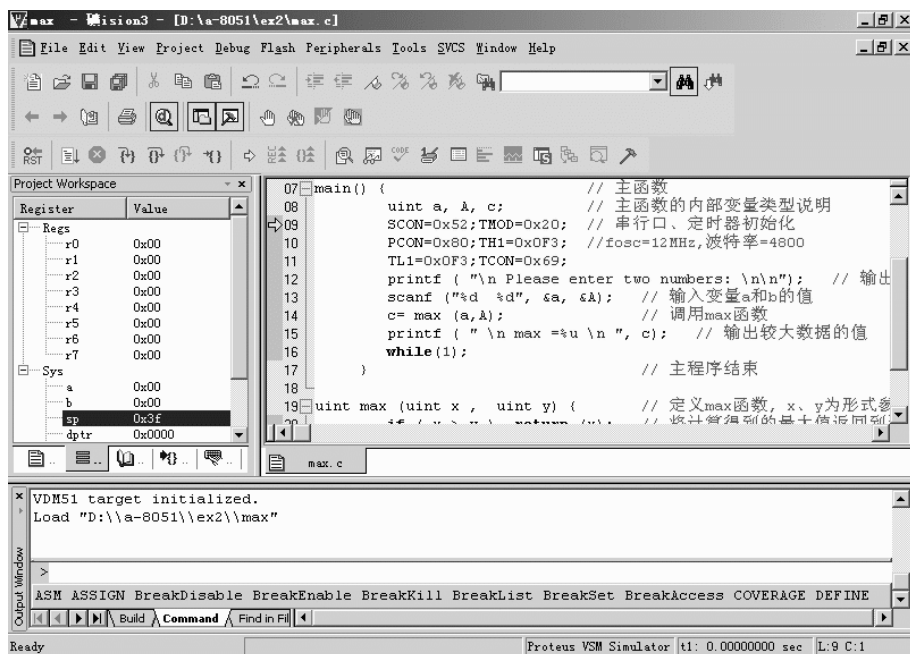


图2.34 准备与Proteus原理图进行联机仿真调试

然后单击 μ Vision3环境的“Debug菜单 / Start / Stop Debug Session”选项，启动 μ Vision3与Proteus联机。联机成功后，自动装入目标代码并运行到main()函数处，项目窗口切换到“Regs”标签页，显示调试过程中单片机内部工作寄存器R0~R7、累加器A、堆栈指针Sp、数据指针寄存器DPTR、程序计数器PC及程序状态字寄存器PSW等的值，如图2.35所示。

图2.35 与Proteus联机成功后的 μ Vision3窗口

在联机仿真状态下，可以直接在 μ Vision3环境中进行程序调试，同时通过Proteus原理图观察程序运行结果。对于本例而言，程序中采用scanf()和printf()函数所进行的输入和输出操作是通过单片机串行口实现的，为此在Proteus原理图中，8051单片机的串行端口引脚上连接了一个虚拟终端，用以观察运行结果。单击 μ Vision3环境的“Debug菜单/Go”选项，启动程序全速运行，然后进入Proteus原理图，单击ISIS环境的“Debug菜单/Virtual Terminal”选项，打开虚拟终端窗口，将鼠标指向该窗口并键入两个数字123和456后回车，即可看到程序运行结果，如图2.36所示。

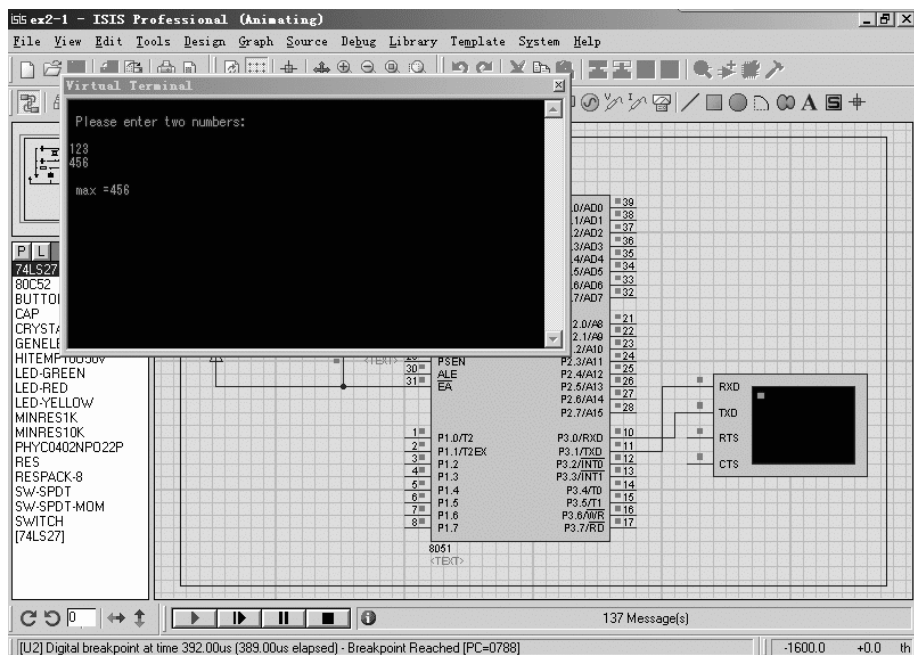


图2.36 在Proteus原理图中观察程序运行结果

μ Vision3与Proteus原理图联机仿真调试的功能十分完善，除了全速运行之外，还可以进行单步、设置断点、运行到光标指定位置等多种操作，调试过程中可同时在 μ Vision3环境与ISIS环境中观察局部变量及用户设置的观察点状态、存储器状态、片内集成外围功能状态等，非常方便。

Keil C51应用程序设计

3.1 Keil C51程序设计的基本语法

Keil C51是一种专为8051单片机设计的高级语言C编译器，支持符合ANSI标准的C语言进行程序设计，同时针对8051单片机自身特点做了一些特殊扩展。C语言对语法的限制不太严格，用户在编写程序时有较大的自由，但它毕竟还是一种程序设计语言，与其他计算机语言一样，采用C语言进行程序设计时，仍需要遵从一定的语法规则。

3.1.1 Keil C51程序的一般结构

与标准C语言相同，C51程序由一个或多个函数构成，其中至少应包含一个主函数main。程序执行时，一定是从main函数开始，调用其他函数后又返回main到函数，被调函数如果位于主调函数前面，则可以直接调用，否则要先说明后调用。这里函数与汇编语言中的子程序类似，函数之间也可以互相调用。C51程序的一般结构为

```
预处理命令          /* 用于包含头文件等 */
全局变量说明        /* 全局变量可被本程序的所有函数引用 */
函数1说明
.....
函数n说明

/* 主函数 */
main() {
    局部变量说明;      /* 局部变量只能在所定义的函数内部引用 */
    执行语句;
    函数调用（形式参数表）;
}

/* 其他函数定义 */
函数1（形式参数说明） {
```

```

    局部变量说明;      /* 局部变量只能在所定义的函数内部引用 */
    执行语句;
    函数调用 (形式参数表);
}
.....
函数n (形式参数说明) {
    局部变量说明;      /* 局部变量只能在所定义的函数内部引用 */
    执行语句;
    函数调用 (形式参数表);
}

```

由此可见，C51程序是由函数所组成的，函数之间可以相互调用，但main()函数只能调用其他功能函数，不能被其他函数调用。其他功能函数可以是C51编译器提供的库函数，也可以由用户按实际需要自行编写。不管main()函数处于程序中的什么位置，程序总是从main()函数开始执行。编写C51程序时要注意以下几点：

- 函数以花括号“{”开始，以花括号“}”结束，包含在“{ }”以内的部分被称为函数体。花括号必须成对出现。如果一个函数内有多对花括号，则最外层花括号为函数体的范围。为使程序增加可读性，便于理解，可以采用缩进方式书写。
- C51程序没有行号，书写格式自由，一行内可以书写多条语句，一条语句也可以分写在多行上。
- 每条语句最后必须以一个分号“;”结尾，分号是C51程序的必要组成部分。
- 每个变量必须先定义后引用。在函数内部定义的变量为局部变量，又称为内部变量，只有定义它的那个函数之内才能够使用。在函数外部定义的变量为全局变量，又称为外部变量，在定义它的那个程序文件中的函数都可以使用它。
- 对程序语句的注释必须放在双斜杠“//”之后，或者放在“/*.....*/”之内。

3.1.2 数据类型

C51数据类型可分为基本数据类型和复杂数据类型。复杂数据类型由基本数据类型构造而成。基本数据类型有char（字符型）、int（整型）、long（长整型）、float（浮点型）、*（指针型）。Keil C51编译器除了支持以上基本数据类型之外，还支持以下扩充数据类型。

- ① bit位类型。可定义一个位变量，但不能定义位指针，也不能定义位数组。
- ② sfr特殊功能寄存器。可以定义8051单片机的所有内部8位特殊功能寄存器。sfr型数据占用一个内存单元，其取值范围为0~255。
- ③ sfr16 16位特殊功能寄存器。它占用两个内存单元，取值范围为0~65535，可以定义8051单片机内部16位特殊功能寄存器。
- ④ sbit可寻址位。可以定义8051单片机内部RAM中的可寻址位或特殊功能寄存器中的可寻址位。

例如，采用如下语句可以将8051单片机P0口地址定义为80H，将P0.1位定义为FLAG1。

```

sfr P0=80H;
sbit FLAG1=P0^1;

```

表3.1为Keil C51编译器能够识别的数据类型。

表3.1 Keil C51编译器能够识别的数据类型

数据类型	长度	值域
unsigned char	单字节	0~255
signed char	单字节	-128~127
unsigned int	双字节	0~65536
signed int	双字节	-32768~32767
unsigned long	四字节	0~4294967295
signed long	四字节	-2147483648~2147483647
float	四字节	$\pm 1.175494E-38 \sim \pm 3.402823E+38$
*	1~3 字节	对象的地址
bit	位	0 或 1
sfr	单字节	0~255
sfr16	双字节	0~65536
sbit	可寻址位	0 或 1

3.1.3 常量、变量及其存储模式

常量包括整型常量（就是整型常数）、浮点型常量（有十进制表示形式和指数表示形式）、字符型常量（单引号内的字符，如 ‘a’）及字符串常量（双引号内的单个或多个字符，如 “a”，“Hello”）等。

变量是一种在程序执行过程中其值能不断变化的量。使用一个变量之前，必须先进行定义，用一个标识符作为变量名并指出它的数据类型和存储模式，以便编译系统为它分配相应的存储单元。在C51 中对变量进行定义的格式为

[存储种类] 数据类型 [存储器类型] 变量名表；

其中，“存储种类”和“存储器类型”是可选项。变量的存储种类有四种：自动（auto）、外部（extern）、静态（static）和寄存器（register）。定义变量时，如果省略存储种类选项，则该变量将为自动（auto）变量。定义变量时，除了需要说明其数据类型之外，Keil C51编译器还允许说明变量的存储器类型，对于每个变量可以根据其存储器的类型确定其存储空间，使其能够在8051单片机系统内进行准确的地址定位。

表3.2为Keil C51编译器所能识别的存储器类型及其地址空间。

表3.2 Keil C51编译器所能识别的存储器类型及其地址空间

存储器类型	说 明
data	低 128 字节片内 RAM，DATA 区（00H~7FH 地址空间），访问速度最快
bdata	可位寻址片内 RAM，BDATA 区（20H~2FH 地址空间），允许位与字节混合访问
idata	256 字节片内 RAM，IDATA 区（00H~FFH 地址空间），允许访问全部片内地址
pdata	分页寻址片外 RAM，PDATA 区（0000~FFFFH 地址空间），用 MOVX @Ri 指令访问
xdata	片外 RAM，XDATA 区（0000~FFFFH 地址空间），用 MOVX @DPTR 指令访问
code	ROM，CODE 区（0000~FFFFH 地址空间），用 MOVC @A+DPTR 指令访问

下面是一些变量定义的例子。

```

char data var1;          /* 在DATA区定义字符型变量var1 */
int idata var2;          /* 在IDATA区定义整型变量var2 */
char code text[]="ENTER PARAMETER: "; /* 在CODE区定义字符串数组text[] */
long xdata array [100]; /* 在XDATA区定义长整型数组变量array[100] */
extern float idata x,y,z; /* 在IDATA 区定义外部浮点型变量 x, y, z */
char bdata flags;        /* 在BDATA 区定义字符型变量flags */
sbit flag0=flags^0;      /* 在BDATA 区定义可位寻址变量flag0 */
sfr P0=0x80;             /* 定义特殊功能寄存器P0 */

```

定义变量时，如果省略“存储器类型”选项，则按编译时所使用的存储器模式SMALL、COMPACT或LARGE来规定默认存储器类型，确定变量的存储器空间，函数中不能采用寄存器传递的参数变量也保存在默认的存储器空间。表3.3为Keil C51编译器不同编译模式对应的存储器类型。

表3.3 Keil C51编译器不同编译模式对应的存储器类型

编译模式	存储器类型
SMALL	DATA
COMPACT	PDATA
LARGE	XDATA

3.1.4 运算符与表达式

Keil C51对数据有很强的表达能力，具有十分丰富的运算符。运算符就是完成某种特定运算的符号，表达式则是由运算符及运算对象所组成的具有特定含义的一个式子。在任意一个表达式的后面加一个分号“;”就构成了一个表达式语句。由运算符和表达式可以组成C51程序的各种语句。

运算符按其在表达式中所起的作用可分为赋值运算符、算术运算符、增量与减量运算符、关系运算符、逻辑运算符、位运算符、复合赋值运算符、逗号运算符、条件运算符、指针和地址运算符、强制类型转换运算符等。

(1) 赋值运算符

在C51程序中，符号“=”被称为赋值运算符。它的作用是将一个数据的值赋给一个变量，利用赋值运算符将一个变量与一个表达式连接起来的式子被称为赋值表达式，在赋值表达式的后面加一个分号“;”便构成了赋值语句。在使用赋值运算符“=”时应注意，不要与关系运算符“==”相混淆。

(2) 算术运算符

C语言中的算术运算符有+（加或取正值）运算符、-（减或取负值）运算符、*（乘）运算符、/（除）运算符、%（取余）运算符。

这些运算符对于加、减和乘法符合一般的算术运算规则，除法运算有所不同。如果是两个整数相除，其结果为整数，则舍去小数部分。如果是两个浮点数相除，则结果为浮点数。取余运算要求两个运算对象均为整型数据。

算术运算符将运算对象连接起来的式子即为算术表达式。在求一个算术表达式的值时，要按运算符的优先级别进行。算术运算符中取负值（-）的优先级最高，其次是乘法（*）、除法（/）和取余（%）运算符，加法（+）和减法（-）运算符的优先级最低。需要时可在算术表达式中采用圆括号来改变运算符的优先级，括号的优先级最高。

（3）增量和减量运算符

C51中除了基本的加、减、乘、除运算符之外，还提供一种特殊的运算符，即++（增量）运算符、--（减量）运算符。

增量和减量是C51中特有的一种运算符。它们的作用分别是对运算对象做加1和减1运算。增量运算符和减量运算符只能用于变量，不能用于常数或表达式，在使用中要注意运算符的位置。例如，++i与i++的意义完全不同，前者为在使用i之前先使i加1，而后者则是在使用i之后再使i的值加1。

（4）关系运算符

C语言中有6种关系运算符，即>（大于）、<（小于）、>=（大于等于）、<=（小于等于）、==（等于）、!=（不等于）。

前4种关系运算符具有相同的优先级，后两种关系运算符也具有相同的优先级。但前4种的优先级高于后两种。用关系运算符将两个表达式连接起来即成为关系表达式。

（5）逻辑运算符

C51中有3种逻辑运算符，即||（逻辑或）、&&（逻辑与）、！（逻辑非）。

逻辑运算符用来求某个条件式的逻辑值，用逻辑运算符将关系表达式或逻辑量连接起来就是逻辑表达式。

关系运算符和逻辑运算符通常用来判别某个条件是否满足。关系运算和逻辑运算的结果只有0和1两种值。当所指定的条件满足时结果为1，条件不满足时结果为0。

上面几种运算符的优先级为（由高至低）逻辑非→算术运算符→关系运算符→逻辑与→逻辑或。

（6）位运算符

C51中共有6种位运算符，即~（按位取反）、<<（左移）、>>（右移）、&（按位与）、^（按位异或）、|（按位或）。

位运算符的作用是按位对变量进行运算，并不改变参与运算变量的值。若希望按位改变运算变量的值，则应利用相应的赋值运算。例如，先用赋值语句a=0xEA将变量a赋值为0xEA，接着对变量a进行移位操作a<<2，其结果是将十六进制数0xEA左移2位，移空的2位补0，移出的2位丢弃，移位的结果为0xA8，而变量a的值在执行后仍为0xEA！如果希望变量a在执行之后为移位操作的结果，则应采用语句a=a<<2。另外，位运算符不能用来对浮点型数据进行操作。位运算符的优先级从高到低依次是按位取反（~）→左移（<<）和右移（>>）→按位与（&）→按位异或（^）→按位或（|）。

（7）复合赋值运算符

在赋值运算符“=”的前面加上其他运算符就构成了所谓复合赋值运算符。C51中共有10种复合赋值运算符，即+=（加法赋值）、-=（减法赋值）、*=（乘法赋值）、/=（除法赋

值)、%= (取模赋值)、<<= (左移位赋值)、>>= (右移位赋值)、&= (逻辑与赋值)、|= (逻辑或赋值)、^= (逻辑异或赋值)、~= (逻辑非赋值)。

复合赋值运算首先对变量进行某种运算,然后将运算的结果再赋给该变量。采用复合赋值运算符,可以使程序简化,同时还可以提高程序的编译效率。

(8) 逗号运算符

在C51中,程序逗号“,”是一个特殊的运算符,可以用它将两个(或多个)表达式连接起来,称为逗号表达式。程序运行时对于逗号表达式的处理,是从左至右依次计算出各个表达式的值,而整个逗号表达式的值是最右边表达式(即表达式 n)的值。

在许多情况下,使用逗号表达式的目的只是为了分别得到各个表达式的值,而并不一定要得到和使用整个逗号表达式的值。另外还要注意,并不是在程序的任何地方出现的逗号,都可以认为是逗号运算符。有些函数中的参数也是用逗号来间隔的,如库输出函数printf(“n%d %d %d”,a,b,c)中的“a,b,c”是函数的三个参数,而不是一个逗号表达式。

(9) 条件运算符

条件运算符“?:”是C51中唯一的一个三目运算符。它要求有三个运算对象,用它可以将三个表达式连接构成一个条件表达式。条件表达式的一般形式为

逻辑表达式 ? 表达式1 : 表达式2

其功能是首先计算逻辑表达式,当值为真(非0值)时,将表达式1的值作为整个条件表达式的值;当逻辑表达式的值为假(0值)时,将表达式2的值作为整个条件表达式的值。例如,条件表达式max=(a>b)?a:b的执行结果是将a和b中较大者赋值给变量max。另外,条件表达式中逻辑表达式的类型可以与表达式1和表达式2的类型不一样。

(10) 指针和地址运算符

指针是C51中的一个十分重要的概念。C51中专门规定了一种指针类型的数据。变量的指针就是该变量的地址,还可以定义一个指向某个变量的指针变量。为了表示指针变量和它所指向的变量地址之间的关系,C51提供了两个专门的运算符:

*(取内容)、&(取地址)。

取内容和取地址运算的一般形式分别为

变量 = * 指针变量

指针变量 = & 目标变量

取内容运算的含义是将指针变量所指向的目标变量的值赋给左边的变量;取地址运算的含义是将目标变量的地址赋给左边的变量。需要注意的是,指针变量中只能存放地址(即指针型数据),不要将一个非指针类型的数据赋值给一个指针变量。例如,下面的语句完成对指针变量赋值(地址值):

```
char data *p; /* 定义指针变量 */
p = 30H;      /* 给指针变量赋值, 30H为8051单片机片内RAM地址 */
```

(11) C51对存储器和特殊功能寄存器的访问

虽然可以采用指针变量来对存储器地址进行操作,但是由于8051单片机存储器结构自身的特点,仅用指针方式访问有时会感觉不太方便,所以C51提供了另外一种访问方法,即利用库函数中的绝对地址访问头文件“absacc.h”来访问不同区域的存储器及片外扩展I/O端

口。在“absacc.h”头文件中进行了如下宏定义：

```
CBYTE[地址] (访问CODE区char型)
DBYTE[地址] (访问DATA区char型)
PBYTE[地址] (访问PDATA区或I/O端口char型)
XBYTE[地址] (访问XDATA区或I/O端口char型)
CWORD[地址] (访问CODE区int型)
DWORD[地址] (访问DATA区int型)
PWORD[地址] (访问PDATA区或I/O端口int型)
XWORD[地址] (访问XDATA区或I/O端口int型)
```

下面语句完成向片外扩展端口地址7FFFH写入一个字符型数据：

```
XBYTE[0x7FFF] = 0x80;
```

下面语句将int型数据0x9988送入外部RAM单元0000H和0001H：

```
XWORD[0] = 0x9988;
```

如果采用如下语句定义一个D/A转换器端口地址

```
#define DAC0832 XBYTE[0x7FFF];
```

那么程序文件中所有出现DAC0832的地方，就是对地址为0x7FFFH的外部RAM单元或I/O端口进行访问。

8051单片机具有100多个品种，为了方便访问不同品种单片机内部特殊功能寄存器，C51提供了多个相关头文件，如reg51.h、reg52.h等，在头文件中对单片机内部特殊功能寄存器及其有位名称的可寻址位进行了定义，编程时只要根据所采用的单片机，在程序文件开始处用文件包含处理命令“#include”将相关头文件包含进来，然后就可以直接引用特殊功能寄存器（注意必须采用大写字母），如下面语句完成的8051定时方式寄存器TMOD的赋值为

```
#include <reg51.h>
TMOD = 0x20;
```

（12）强制类型转换运算符

C语言中的圆括号“()”也可作为一种运算符使用。这就是强制类型转换运算符。它的作用是将表达式或变量的类型强制转换成为所指定的类型。在C51程序中进行算术运算时，需要注意数据类型的转换，数据类型转换分为隐式转换和显式转换。隐式转换是在对程序进行编译时由编译器自动处理的，并且只有基本数据类型（即char、int、long和float）可以进行隐式转换。其他数据类型不能进行隐式转换，如我们不能把一个整型数利用隐式转换赋值给一个指针变量，在这种情况下就必须利用强制类型转换运算符来进行显式转换。强制类型转换运算符的一般使用形式为

(类型) = 表达式

显式强制类型转换在给指针变量赋值时特别有用。例如，预先在8051单片机的片外数据存储区（xdata）中定义了一个字符型指针变量px，如果想给这个指针变量赋一初值0xB000，则可以写成px=(char xdata *)0xB000。这种方法特别适合于用标识符来存取绝对地址。

3.2 C51程序的基本语句

3.2.1 表达式语句

C51提供了十分丰富的程序控制语句。表达式语句是最基本的一种语句。在表达式的后边加一个分号“;”就构成了表达式语句。表达式语句也可以仅由一个分号“;”组成。这种语句称为空语句。空语句在程序设计中有时是很有用的,当程序在语法上需要有一个语句,但在语义上并不要求有具体的动作时,便可以采用空语句。

3.2.2 复合语句

复合语句是由若干条语句组合而成的一种语句。它是用一个大括号“{”将若干条语句组合在一起而形成的一种功能块。复合语句不需要以分号“;”结束,但它内部的各条单语句仍需以分号“;”结束。复合语句的一般形式为

```
{  
    局部变量定义;  
    语句1;  
    语句2;  
    .....  
    语句n;  
}
```

复合语句在执行时,其中各条单语句依次顺序执行。整个复合语句在语法上等价于一条单语句。复合语句允许嵌套,即在复合语句内部还可以包含别的复合语句。通常复合语句出现在函数中。实际上,函数的执行部分(即函数体)就是一个复合语句。复合语句中的单语句一般是可执行语句,也可以是变量定义语句。在复合语句内所定义的变量,称为该复合语句中的局部变量,它仅在当前这个复合语句中有效。

3.2.3 条件语句

条件语句又称为分支语句。它是用关键字“if”构成的。C51提供了三种形式的条件语句。

(1)

```
if (条件表达式) 语句
```

其含义为:若条件表达式的结果为真(非0值),就执行后面的语句;反之,若条件表达式的结果为假(0值),就不执行后面的语句。这里的语句也可以是复合语句。

(2)

```
if (条件表达式) 语句1  
else 语句2
```

其含义为:若条件表达式的结果为真(非0值),就执行语句1;反之,若条件表达式的

结果为假（0值），就执行语句2。这里的语句1和语句2均可以是复合语句。

(3)

```
if (条件表达式1)    语句1
else if(条件式表达2) 语句2
else if(条件式表达3) 语句3
...
else if(条件表达式n) 语句m
else                语句n
```

这种条件语句常用来实现多方向条件分支。

3.2.4 开关语句

开关语句也是一种用来实现多方向条件分支的语句。虽然采用条件语句也可以实现多方向条件分支，但是当分支较多时会使条件语句的嵌套层次太多，程序冗长，可读性降低。开关语句直接处理多分支选择，使程序结构清晰，使用方便。开关语句是用关键字switch 构成的，一般形式为：

```
switch (表达式)
{
    case 常量表达式1 : 语句1;
                        break;
    case 常量表达式2 : 语句2;
                        break;
    ...
    case 常量表达式n : 语句n;
                        break;
    default: 语句d
}
```

开关语句的执行过程是：将switch后面表达式的值与case后面各个常量表达式的值逐个进行比较，若遇到匹配时，就执行相应case后面的语句，然后执行break语句，break语句又称中断语句，它的功能是中止当前语句的执行，使程序跳出switch语句。若无匹配的情况，则只执行语句d。

3.2.5 循环语句

在实际应用中有很多地方需要用到循环控制，如对于某种操作需要反复进行多次等。在C51程序中用来构成循环控制的语句有while语句、do-while语句、for语句及goto语句，分述如下。

采用while语句构成循环结构的一般形式为

```
while (条件表达式) 语句;
```

其意义为，当条件表达式的结果为真（非0值）时，程序就重复执行后面的语句，一直执行到条件表达式的结果变为假（0值）时为止。这种循环结构是先检查条件表达式所给出

的条件，再根据检查的结果决定是否执行后面的语句。如果条件表达式的结果一开始就为假，则后面的语句一次也不会被执行。这里的语句可以是复合语句。

采用do-while 语句构成循环结构的一般形式为

```
do 语句 while(条件表达式);
```

这种循环结构的特点是先执行给定的循环体语句，再检查条件表达式的结果。当条件表达式的值为真（非0值）时，则重复执行循环体语句，直到条件表达式的值变为假（0值）时为止。因此，用do-while语句构成的循环结构在任何条件下，循环体语句至少会被执行一次。

采用for语句构成循环结构的一般形式为：

```
for ([初值设定表达式]; [循环条件表达式]; [更新表达式]) 语句
```

for语句的执行过程是：先计算出初值设定表达式的值作为循环控制变量的初值，再检查循环条件表达式的结果，当满足条件时，就执行循环体语句并计算更新表达式，再根据更新表达式的计算结果判断循环条件是否满足……一直进行到循环条件表达式的结果为假（0值）时退出循环体。

3.2.6 goto、break、continue语句

goto语句是一个无条件转向语句，它的一般形式为

```
goto 语句标号;
```

其中，语句标号是一个带冒号“:”的标识符。将goto语句和if语句一起使用，可以构成一个循环结构。但更常见的是在C51程序中，采用goto语句来跳出多重循环。需要注意的是，只能用goto语句从内层循环跳到外层循环，而不允许从外层循环跳到内层循环。

break语句也可以用于跳出循环语句，它的一般形式为

```
break;
```

对于多重循环的情况，break语句只能跳出它所处的那一层循环，而不像goto语句可以直接从最内层循环中跳出来。由此可见，要退出多重循环时，采用goto语句比较方便。需要指出的是，break语句只能用于开关语句和循环语句之中，是一种具有特殊功能的无条件转移语句。

Continue是一种中断语句，它的功能是中断本次循环，一般形式为

```
Continue;
```

continue语句通常和条件语句一起用在由while、do—while和for语句构成的循环结构中，也是一种具有特殊功能的无条件转移语句，但与break语句不同，continue语句并不跳出循环体，而只是根据循环控制条件确定是否继续执行循环语句。

3.2.7 返回语句

返回语句用于终止函数的执行，并控制程序返回到调用该函数时所处的位置。返回语句有两种形式：

① return (表达式);

② return;

如果return语句后边带有表达式,则要计算表达式的值,并将表达式的值作为该函数的返回值。若使用不带表达式的第2种形式,则被调用函数返回主调函数时,函数值不确定。一个函数的内部可以含有多个return语句,但程序仅执行其中的一个return语句而返回主调函数。一个函数的内部也可以没有return语句,在这种情况下,当程序执行到最后一个界限符“}”处时,就自动返回主调函数。

➔ 3.3 函数

3.3.1 函数的定义与调用

从用户的角度来看有两种函数:标准库函数和用户自定义函数。标准库函数是 Keil C51 编译器提供的,不需要用户进行定义,可以直接调用。用户自定义函数是用户根据自己需要编写的能实现特定功能的函数,它必须先进行定义之后才能调用。函数定义的一般形式为

```
函数类型  函数名 (形式参数表)
{
    局部变量定义
    函数体语句
}
```

其中,“函数类型”说明了自定义函数返回值的类型。

“函数名”是用标识符表示的自定义函数名字。

“形式参数表”中列出的是在主调用函数与被调用函数之间传递数据的形式参数,形式参数的类型必须要加以说明。ANSI C标准允许在形式参数表中对形式参数的类型进行说明。如果定义的是无参函数,则可以没有形式参数表,但圆括号不能省略。

“局部变量定义”是对在函数内部使用的局部变量进行定义。

“函数体语句”是为完成该函数的特定功能而设置的各种语句。

C51程序中函数是可以互相调用的。所谓函数调用就是在一个函数体中引用另外一个已经定义了的函数,前者被称为主调函数,后者被称为被调用函数。函数调用的一般形式为

```
函数名 (实际参数表)
```

其中,“函数名”指出被调用的函数。

“实际参数表”中可以包含多个实际参数,各个参数之间用逗号隔开。实际参数的作用是将它的值传递给被调用函数中的形式参数。需要注意的是,函数调用中的实际参数与函数定义中的形式参数必须在个数、类型及顺序上严格保持一致,以便将实际参数的值正确地传递给形式参数。否则,在函数调用时会产生意想不到的结果。如果调用的是无参函数,则可以没有实际参数表,但圆括号不能省略。

在C51中可以采用三种方式完成函数的调用。

① 函数语句。在主调函数中将函数调用作为一条语句。这是无参调用,它不要求被调函数返回一个确定的值,只要求它完成一定的操作。

② 函数表达式。在主调函数中将函数调用作为一个运算对象直接出现在表达式中，这种表达式被称为函数表达式。这种函数调用方式通常要求被调函数返回一个确定的值。

③ 函数参数。在主调函数中将函数调用作为另一个函数调用的实际参数。这种在调用一个函数的过程中又调用了另外一个函数的方式被称为嵌套函数调用。

与使用变量一样，在调用一个函数之前（包括标准库函数），必须对该函数的类型进行说明，即“先说明，后调用”。如果调用的是库函数，则一般应在程序的开始处用预处理命令#include 将有关函数说明的头文件包含进来。

如果调用的是用户自定义函数，而且该函数与调用它的主调函数在同一个文件中，则一般应该在主调函数中对被调函数的类型进行说明。函数说明的一般形式为

类型标识符 被调用的函数名(形式参数表);

其中，“类型标识符”说明了函数返回值的类型。

“形式参数表”中说明各个形式参数的类型。

需要注意的是，函数的定义与函数的说明是完全不同的，二者在书写形式上也不一样。函数定义时，被定义函数名的圆括号后面没有分号“;”，即函数定义还未结束，后面应接着写被定义的函数体部分。而函数说明结束时，在圆括号的后面需要有一个分号“;”作为结束标志。

3.3.2 中断服务函数与寄存器组定义

C51编译器支持在C语言源程序中直接编写8051单片机的中断服务函数程序，一般形式为

函数类型 函数名(形式参数表) [interrupt n] [using m]

关键字interrupt后面的 n 是中断号， n 的取值范围为0~31。编译器从 $8n+3$ 处产生中断向量，具体的中断号 n 和中断向量取决于8051系列单片机芯片型号。常用的中断源和中断向量见表3.4。

表3.4 常用的中断源和中断向量

中断号 n	中 断 源	中断向量 $8n+3$
0	外部中断 0	0003H
1	定时器 0	000BH
2	外部中断 1	0013H
3	定时器 1	001BH
4	串行口	0023H

8051系列单片机可以在片内RAM中使用4个不同的工作寄存器组，每个寄存器组中包含8个工作寄存器（R0~R7）。C51编译器扩展了一个关键字using，专门用来选择8051单片机中不同的工作寄存器组。using后面的 m 是一个0~3的常整数，分别选中4个不同的工作寄存器组。在定义一个函数时，using是一个选项，如果不用该选项，则由编译器自动选择一个寄存器组做绝对寄存器组访问。

编写8051单片机中断函数时应遵循以下规则:

- 中断函数不能进行参数传递, 也没有返回值。因此建议在定义中断函数时将其定义为void类型, 以明确说明没有返回值。
- 在任何情况下都不能直接调用中断函数, 否则会产生编译错误。
- 如果在中断函数中调用了其他函数, 则被调用函数所使用的寄存器组必须与中断函数相同, 否则会产生不正确的结果, 这一点必须引起足够的注意。
- C51编译器从绝对地址 $8n+3$ 处产生一个中断向量, 其中 n 为中断号。该向量包含一个到中断函数入口地址的绝对跳转。

3.4 Keil C51编译器对ANSI C的扩展

3.4.1 存储器类型与编译模式

8051单片机的存储器空间可分为三种: 片内、外统一编址的程序存储器ROM, 片内数据存储器RAM和片外数据存储器RAM。

C51编译器对于ROM存储器提供存储器类型标识符code、用户的应用程序代码及各种表格常数定位在CODE空间。

数据存储器RAM用于存放各种变量, 通常应尽可能将变量放在片内RAM中以加快操作速度。C51编译器对片内RAM提供三种存储器类型标识符: data、idata和bdata。data地址范围为 $0x00 \sim 0x7f$, 位于DATA空间的变量以直接寻址方式操作, 速度最快; idata地址范围为 $0x00 \sim 0xff$, 位于IDATA空间的变量以寄存器间接寻址方式操作, 速度略慢于DATA空间; bdata地址范围为 $0x20 \sim 0x2f$, 位于BDATA空间的变量除了可以进行直接寻址或间接寻址操作之外, 还可以进行位寻址操作。

片外数据RAM简称XRAM。C51提供两个存储器类型标识符: xdata和pdata。xdata地址范围为 $0x0000 \sim 0xffff$, 位于XDATA空间的变量以MOVX @DPTR方式寻址, 可以操作整个64KB地址范围内的变量, 但这种方式速度最慢, PDATA空间又被称为片外分页XRAM空间, 它将地址 $0x0000 \sim 0xffff$ 均匀地分成256页, 每页的地址都为 $0x00 \sim 0xff$, 位于PDATA空间的变量以MOVX @R0、MOVX @R1方式寻址。实际上, XRAM空间并非全部用于存放变量, 用户扩展的I/O接口也位于XRAM地址范围之内。有些新型的8051单片机还提供片内XRAM。其操作方式与传统XRAM相同, 但一般要先对相应的特殊功能寄存器SFR进行配置之后才能使用。

一些新型8051单片机能够进行大容量存储器扩展, 如Philips公司的80C51Mx系列可扩展高达8MB的CODE和XDATA存储器空间, Dallas公司的80C390系列和Analog公司的Aduc8xx系列采用24位的数据指针DPTR以邻接方式可扩展高达16MB的CODE和XDATA存储器空间。C51编译器针对这种大容量扩展存储器定义了far和const far两种存储器类型, 分别用以操作这种扩展的片外RAM和片外ROM空间。对于传统的8051单片机, 如果它具有可以映像到XDATA的附加存储器空间, 或者提供了一种地址扩展特殊功能寄存器(address extension SFR), 则可以根据具体硬件电路通过修改配置文件XBANKING.A51来使用far和const far类型

的变量。需要注意的是,在使用far和const far存储器类型时,必须采用LX51扩展连接定位器,同时还必须采用OMF2格式的目标文件。

表3.5为Keil C51编译器能够识别的存储器类型,定义变量时,可以采用上述存储器类型明确指出变量的存储器空间。

表3.5 Keil C51编译器能够识别的存储器类型

存储器类型	说 明
code	程序存储器(64K字节),用MOVC @A+DPTR指令访问
data	直接寻址的片内数据存储器(128字节),访问速度最快
idata	片内数据存储器(256字节),允许访问全部片内RAM地址
bdata	可位寻址的片内数据存储器(16字节),允许位与字节混合访问
xdata	片外数据存储器(64K字节),用MOVX @DPTR指令访问
pdata	分页寻址的片外数据存储器(256字节),用MOVX @R0, MOVX @R1指令访问
far	高达16MB的扩展RAM和ROM,专用芯片扩展访问(Philips 80C51Mx,DS80C390)或用户自定义子程序进行访问

如果定义变量时没有明确指出具体的存储器类型,则按C51编译器采用的编译模式来确定变量的默认存储器空间。Keil C51编译器控制命令SMALL、COMPACT、LARGE对变量存储器空间的影响如下。

① SMALL。所有变量都定义在8051单片机的片内RAM中,对这种变量的访问速度最快。另外,堆栈也必须位于片内RAM中,而堆栈的长度是很重要的,实际栈长取决于不同函数的嵌套深度。采用SMALL编译模式与定义变量时指定data存储器类型具有相同效果。

② COMPACT。所有变量定义在分页寻址的片外XRAM中,每一页片外XRAM的长度为256字节。这时对变量的访问是通过寄存器间接寻址(MOVX @R0, MOVX @R1)进行的,变量的低8位地址由R0或R1确定,变量的高8位地址由P2口确定。采用这种模式时,必须适当改变配置文件STARTUP.A51中的参数PDATASTART和PDATALEN。同时还必须在μVision2的“Options选项/BL51 Locator标签栏/Pdata框”中输入合适的地址参数,以确保P2口能输出所需要的高8位地址。采用COMPACT编译模式与定义变量时指定pdata存储器类型具有相同效果。

③ LARGE。所有变量定义在片外XRAM中(最大可达64 KB),使用数据指针DPTR来间接访问变量(MOVX @DPTR),这种编译模式对数据访问的效率最低,而且将增加程序的代码长度。采用LARGE编译模式与定义变量时指定xdata存储器类型具有相同效果。

3.4.2 关于bit、sbit、sfr、sfr16数据类型

Keil C51编译器支持标准C语言的数据类型,另外还根据8051单片机的特点扩展了bit、sbit、sfr、sfr16数据类型。

(1) bit

在C51程序中可以定义bit类型的变量、函数、函数参数及返回值,如

```
static bit done_flag = 0;      /* bit类型变量 */
bit testfunc (                /* bit类型函数 */
    bit flag1,                /* bit类型函数参数 */
```

```

    bit flag2)
{
    .....;
    return (0);          /* bit类型返回值 */
}

```

所有bit类型的变量都被定位在8051片内RAM的可位寻址区。由于8051单片机的可位寻址区只有16字节，所以在某个范围内最多只能声明128个bit类型变量。声明bit类型变量时可以带有存储器类型data、idata或bdata。对于bit类型变量有如下限制：如果在函数中采用预处理命令#pragma disable禁止了中断，或者在函数声明时采用了关键字using n明确进行了寄存器组切换，则该函数不能返回bit类型的值，否则C51在进行编译时会产生编译错误；另外不能定义bit类型指针，也不能定义bit类型数组。

(2) sbit

关键字sbit用于定义可独立寻址访问的位变量，简称可位寻址变量。C51编译器提供一个存储器类型bdata，带有bdata存储器类型的变量定位在8051单片机片内RAM的可位寻址区，带有bdata存储器类型的变量可以进行字节寻址也可以进行位寻址，因此对bdata变量可用sbit指定其中任意位为可位寻址变量。需要注意的是，采用bdata和sbit所定义的变量都必须必须是全局变量，并且采用sbit定义可位寻址变量时要求基址对象的存储器类型为bdata。例如，可先定义变量的数据类型和存储器类型为

```

int bdata ibase;          /* 定义ibase为bdata整型变量 */
char bdata bary[4];      /* 定义bary[4]为bdata字符型数组 */

```

然后使用sbit定义可位寻址变量为

```

sbit mybit0 = ibase^0;    /* 定义mybit0为ibase的第0位 */
sbit mybit15 = ibase^15; /* 定义mybit15为ibase的第15位 */
sbit Ary07 = bary[0]^7;  /* 定义Ary07为bary[0]的第7位 */
sbit Ary37 = bary[3]^7;  /* 定义Ary37为bary[3]的第7位 */

```

操作符“^”后面的数值范围取决于基址变量的数据类型，对于char型而言是0~7，对于int型而言是0~15，对于long型是0~31。bdata变量ibase和bdata数组bary[4]可以进行字或字节寻址，sbit变量可以直接操作可寻址位，如

```

ibase = -1;              /* 字寻址，对ibase赋值为-1 */
bary[3] = 'a';          /* 字节寻址，对bary[3]赋值为'a' */
Ary37 = 0;               /* 清0 bary[3]的第7位 */
mybit15 = 1;             /* 置1 ibase的第15位 */

```

对于bdata变量可以向data变量一样处理，所不同的是bdata变量必须位于8051单片机的片内RAM的可位寻址区，其长度不能超过16字节。

sbit还可以用于定义结构与联合，利用这一特点可以实现对float型数据指定bit变量，如

```

union lft {
    float mf;
    long ml;
};

```

```

bdata struct bad {
    char mc;
    union lft u;
} tcp;
sbit tcpf31 = tcp.u.ml ^ 31;          /* float数据的第31位 */
sbit tcpml0 = tcp.mc ^ 0;
sbit tcpml7 = tcp.mc ^ 7;

```

采用sbit类型时，需要指定一个变量作为基地址，再通过指定该基地址变量的bit位置来获得实际的物理bit地址，并不是所有类型变量的物理bit地址都与其逻辑bit地址相一致，物理上的bit 0对应第一个字节的bit 0，物理上的bit 8对应第二个字节的bit 0。对于int类型的数据，由于是按高字节在前的方式存储的，int类型数据的bit 0应位于第二个字节的bit 0，所以采用sbit指定int类型数据bit 0时应使用物理上的bit 8。

(3) sfr

8051单片机片内RAM中与idata空间相重叠的高128字节（地址范围80~FFH）被称为特殊功能寄存器（SFR）区。单片机内部集成功能的操作都是通过特殊功能寄存器来实现的。为了能够直接访问8051系列单片机内部特殊功能寄存器，C51编译器扩充了关键字sfr和sfr16，利用这种扩充关键字可以在C51源程序中直接定义8051单片机的特殊功能寄存器。定义方法为

sfr特殊功能寄存器名 = 地址常数;

例如，

```

sfr P0 = 0x80;          /* 定义P0寄存器，地址为0x80 */
sfr SCON = 0x90;        /* 定义串行口控制寄存器，地址为 0x90 */

```

这里需要注意的是，在关键字sfr后面必须跟一个标识符作为特殊功能寄存器名，名字可任意选取，但应符合一般习惯。等号后面必须是常数，不允许有带运算符的表达式。对于传统8051单片机地址常数的范围是0x80~0xff，对于Philips 80C51Mx单片机，地址常数的范围是0x180~0x1ff。

(4) sfr16

在一些新型8051单片机中，特殊功能寄存器经常组合成16位来使用。采用关键字sfr16可以定义这种16位的特殊功能寄存器。例如，对于8052单片机的定时器T2，可采用如下的方法来定义，即

```

sfr16 T2 = 0xCC;        /* 定义TIMER2，其地址为T2L=0xCC，T2H=0xCD */

```

这里T2为特殊功能寄存器名，等号后面是它的低字节地址，其高字节地址必须在物理上直接位于低字节之后。这种定义方法适用于所有新一代8051单片机中新增加的特殊功能寄存器。

在8051单片机应用系统中经常需要访问特殊功能寄存器中的一些特定位置，可以利用C51编译器提供的扩充关键字sbit定义特殊功能寄存器中的可位寻址对象。定义方法有如下三种。

(1) sbit 位变量名 = 位地址

这种方法将位的绝对地址赋给位变量，位地址必须位于0x80~0xFF之间，如

```
sbit OV = 0xD2;
sbit CY = 0xD7;
```

(2) **sbit** 位变量名 = 特殊功能寄存器名^位位置

当可寻址位位于特殊功能寄存器中时可采用这种方法，“位位置”是一个0~7之间的常数，如

```
sfr PSW = 0xD0;
sbit OV = PSW^2;
sbit CY = PSW^7;
```

(3) **sbit** 位变量名 = 字节地址^位位置

这种方法以一个常数（字节地址）作为基地址，该常数必须在0x80H~0xFF之间。“位位置”是一个0~7之间的常数，如

```
sbit OV = 0xD0^2;
sbit CY = 0xD0^7;
```

需要注意的是，用**sbit**定义的特殊功能寄存器中的可寻址位是一个独立的定义类（class），不能与其他位定义和位域互换。

3.4.3 一般指针与基于存储器的指针及其转换

Keil C51编译器支持两种指针类型：一般指针和基于存储器的指针。一般指针需要占用3个字节，基于存储器的指针只需要1~2个字节。一般指针具有较好的兼容性，但运行速度较慢，基于存储器的指针是C51编译器专门针对8051单片机存储器特点进行的扩展，只适用于8051单片机，但具有较高的运行速度。

定义一般指针的方法与ANSI C相同，如

```
char * sptr;          /* char 型指针 */
int * numptr          /* int 型指针 */
```

一般指针在内存中占用3个字节，第一个字节存放该指针的存储器类型编码（由编译模式确定），第二和第三个字节分别存放该指针的高位和低位地址偏移量。一般指针的存储器类型编码值见表3.6。

表3.6 一般指针的存储器类型编码位

存储器类型 1	idata/data/bdata	xdata	pdata	code
编码值	0x00	0x01	0xFE	0xFF

一般指针可用于存取任何变量而不必考虑变量在8051单片机存储器空间的位置，许多C51库函数采用了一般指针。函数可以利用一般指针来存取位于任何存储器空间的数据。

定义一般指针时可以在“*”号后面带一个“存储器类型”选项，用以指定一般指针本身的存储器空间位置，如

```
char * xdata strptr;    /* 位于 xdata 空间的一般指针 */
int * data numptr;      /* 位于 data 空间的一般指针 */
long * idata varptr;    /* 位于 idata 空间的一般指针 */
```

由于一般指针所指对象的存储器空间位置只有在运行期间才能确定，编译器在编译期间无法优化存储方式，必须生成一般代码以保证能对任意空间的对象进行存取，因此一般指针所产生的代码运行速度较慢，如果希望加快运行速度，则应采用基于存储器的指针。

基于存储器的指针所指对象具有明确的存储器空间，长度可为1个字节（存储器类型为idata、data、pdata）或2个字节（存储器类型为code、xdata）。定义指针时，如果在“*”号前面增加一个“存储器类型”选项，则该指针就被定义为基于存储器的指针，如

```
char data * str;           /* 指向data空间char型数据的指针 */
int xdata * num;          /* 指向xdata空间int型数据的指针 */
long code * pow;          /* 指向code空间long型数据的指针 */
```

与一般指针类似，定义基于存储器的指针时还可以指定指针本身的存储器空间位置，即在“*”号后面带一个“存储器类型”选项，如

```
char data*xdata str; /*指向data空间char型数据的指针，指针本身在xdata空间*/
int xdata*data num; /*指向xdata空间char型数据的指针，指针本身在data空间*/
long code*idata pow; /*指向code空间long型数据的指针，指针本身在idata空间*/
```

基于存储器的指针长度比一般指针短，可以节省存储器空间，运行速度快，但它所指对象具有确定的存储器空间，缺乏兼容性。

一般指针与基于存储器的指针可以相互转换。在某些函数调用中进行参数传递时需要采用一般指针，如C51的库函数printf()、sprintf()、gets()等便是如此，当传递的参数是基于存储器的指针时，若不特别指明，C51编译器会自动将其转换为一般指针。需要注意的是，如果采用基于存储器的指针作为自定义函数的参数，而程序中又没有给出该函数原型，则基于存储器的指针就自动转换为一般指针。假如在调用该函数时的确需要采用基于存储器的指针（其长度较短）作为传递参数，那么指针的自动转换就可能导致错误。为避免这类错误，应该在程序的开始处用预处理命令“#include”将函数原型说明文件包含进来，或者直接给出函数原型声明。

3.4.4 C51编译器对ANSI C函数定义的扩展

（1）C51编译器支持的函数定义一般形式

C51编译器提供了几种对于ANSI C函数定义的扩展，可用于选择函数的编译模式、规定函数所使用的工作寄存器组、定义中断服务函数、指定再入方式等。在C51程序中进行函数定义的一般格式为

```
函数类型  函数名(形式参数表) [编译模式] [reentrant] [interrupt n] [using n]
{  局部变量定义
    函数体语句
}
```

其中，“函数类型”说明了自定义函数返回值的类型。

“函数名”是用标识符表示的自定义函数名字。

“形式参数表”中列出了在主调用函数与被调用函数之间传递数据的形式参数，形式

参数的类型必须要加以说明。如果定义无参函数，则可以没有形式参数表，但圆括号不能省略。

“局部变量定义”是对在函数内部使用的局部变量进行定义。

“函数体语句”是为完成该函数的特定功能而设置的各种语句。

“编译模式”选项是C51对ANSI C的扩展，可以是SMALL、COMPACT或LARGE，用于指定函数中局部变量和参数的存储器空间。

“reentrant”选项是C51对ANSI C的扩展，用于定义再入函数。

“interrupt n”选项是C51对ANSI C的扩展，用于定义中断服务函数，其中“n”为中断号，可为0~31，根据中断号可以决定中断服务程序的入口地址。

“using n”选项是C51对ANSI C的扩展，其中“n”可以是0~3，用于确定中断服务函数所使用的工作寄存器组。

(2) 堆栈及函数的参数传递

函数在运行过程中需要使用堆栈。8051单片机的堆栈必须位于片内RAM空间，其最大范围只有256个字节（对于一些新的扩展型8051单片机，C51编译器可以使用其扩展堆栈区，扩展堆栈区最大可达几千个字节）。为了节省堆栈空间，C51编译器采用一个固定的存储器区域来进行函数参数的传递，发生函数调用时，主调函数先将实际参数复制到该固定的存储器区域，再将程序流程控制交给被调函数，被调函数从该固定的存储器区域取得所需要的参数进行操作。这样就只需要将函数的返回地址保存到堆栈区中。由于中断服务函数可能要进行工作寄存器组切换，因此需要采用较多的堆栈空间。

C51编译器可以采用控制命令“REGPARMS”和“NOREGPARMS”来决定是否通过工作寄存器传递函数参数，在默认状态下，C51编译器可以通过工作寄存器传递最多3个函数参数，这种方式可以提高程序执行效率。如果没有寄存器可用，则通过固定的存储器区域来传递函数的参数。

(3) 函数的编译模式

不同类型8051单片机片内RAM空间大小不同，有些衍生产品只有64个字节的片内RAM，因此在定义函数时要根据具体情况来决定应采用的编译模式，函数参数和局部变量都存放在由编译模式决定的默认存储器空间。可以根据需要对不同函数采用不同的编译模式，在SMALL编译模式下，函数参数和局部变量被存放在8051的片内RAM空间，这种方式对数据的处理效率最高。但片内RAM空间有限，对于较大的程序，若采用SMALL编译模式可能不能满足要求，这时就需要采用其他编译模式。下面的不同函数采用了不同的编译模式。

```
#pragma small                                /* 默认编译模式为SMALL*/
extern int calc (char i, int b) large reentrant; /* 采用LARGE编译模式*/
extern int func (int i, float f) large;         /* 采用LARGE编译模式*/
extern void *tcp (char xdata *xp, int ndx) small; /* 采用SMALL编译模式*/
int mtest (int i, int y){                      /* 采用默认编译模式 */
    return (i * y + y * i + func(-1, 4.75));
}
int large_func (int i, int k)large {            /* 采用Large编译模式 */
    return (mtest (i, k) + 2);
}
```

(4) 寄存器组切换

8051单片机片内RAM中最低32个字节平均分为4个组，每组8个字节都命名为R0~R7，统称为工作寄存器组。这一特点对于编写中断服务函数或使用实时操作系统都十分有用。利用扩展关键字“using”可以在定义函数时规定所使用的工作寄存器组，只要在“using”后面跟一个数字0~3，即可规定所使用的工作寄存器组。

需要注意的是，关键字using不能用在以寄存器返回一个值的函数中，并且要保证任何寄存器组的切换都只在仔细控制的区域内发生，如果不做到这一点，将产生不正确的函数结果。另外，带“using”属性的函数原则上不能返回bit类型的值。

8051单片机复位时，PSW的值为0x00，因此在默认状态下，所有非中断函数都将使用工作寄存器0区。C51编译器可以通过控制命令“REGISTERBANK”为源程序中的所有函数指定一个默认的工作寄存器组，为此用户需要修改启动代码选择不同的寄存器组，然后采用控制命令“REGISTERBANK”来指定新的工作寄存器组。

在默认状态下，C51编译器生成的代码将使用绝对寻址方式来访问工作寄存器R0~R7，从而提高操作性能。绝对寄存器寻址方式可以通过编译控制命令“AREGS”或“NOARGES”来激活或禁止。采用了绝对寄存器的函数不能被另一个使用了不同工作寄存器组的函数所调用，否则会导致不可预知的结果。为了使函数对当前工作寄存器组不敏感，该函数必须采用控制命令“NOARGES”进行编译，这一点对于需要同时从主程序和使用不同寄存器组的中断服务程序中调用的函数时十分有用。

特别需要注意的是，C51编译器对函数之间使用的工作寄存器组是否匹配不做检查，因此使用了交替寄存器组的函数只能调用没有设定默认寄存器组的函数。

(5) 中断函数

利用扩展关键字“interrupt”可以直接在C51程序中定义中断服务函数，“interrupt”跟一个0~31的数字，用于规定中断源和中断入口。关键字“interrupt”对中断函数目标代码的影响如下：

- 在进入中断函数时，特殊功能寄存器 ACC、B、DPH、DPL、PSW将被保存入栈；
- 如果不使用关键字using进行工作寄存器组切换，则将中断函数中所用到的全部工作寄存器都入栈保存；
- 函数退出之前，所有的寄存器内容出栈恢复；
- 中断函数由8051单片机指令RETI结束；
- C51编译器根据中断号自动生成中断函数入口向量地址。

(6) 再入函数

利用C51编译器的扩展关键字“reentrant”可以定义一个再入函数，再入函数可以进行递归调用，或者同时被两个以上其他函数同时调用。通常在实时系统应用中，或中断函数与非中断函数需要共享一个函数时，应将该函数定义为再入函数。

再入函数可被递归调用，无论何时，包括中断服务函数在内的任何函数都可调用再入函数。与非再入函数的参数传递和局部变量的存储分配方法不同。C51编译器为再入函数生成一个模拟栈，通过这个模拟栈来完成参数传递和存放局部变量。根据再入函数所采用的编译模式，模拟栈可以位于片内或片外存储器空间，SMALL模式下再入栈位于data空间，COMPACT模式下再入栈位于pdata空间，LARGE模式下再入栈位于xdata空间。当程序中包

含有多种存储器模式的再入函数时，C51编译器为每种模式单独建立一个模拟栈并独立管理各自的栈指针。再入函数的局部变量及参数都被放在再入栈中，从而使再入函数可以进行递归调用。而非再入函数的局部变量被放在再入栈之外的暂存区内，如果对非再入函数进行递归调用，则上次调用时使用的局部变量数据将被覆盖。

Keil C51编译器对于再入函数有如下规定：

- 再入函数不能传送bit类型的参数，也不能定义局部位变量，再入函数不能操作可位寻址变量；
- 与PL/M51兼容的alien函数不能具有reentrant属性，也不能调用再入函数；
- 再入函数可以同时具有其他属性，如“interrupt”、“using”等，还可以明确声明其存储器模式（SMALL、COMPACT、LARGE）；
- 在同一个程序中可以定义和使用不同存储器模式的再入函数，每个再入函数都必须具有合适的函数原型，原型中还应包含该函数的存储器模式；
- 再入函数的返回地址保存在8051单片机的硬件堆栈内，任意其他的PUSH和POP指令都会影响8051硬件堆栈；
- 不同存储器模式下的再入函数具有其自己的模拟再入栈和再入栈指针，若在同一个模块内定义了SMALL和LARGE模式的再入函数，则C51编译器会同时生成对应的两种再入栈及其再入栈指针。

8051单片机的常规栈总是位于内部数据RAM中而且是“向上生长”型的，而模拟再入栈是“向下生长”型的，如果编译时采用SMALL模式，则常规栈和再入函数的模拟栈将被都放在内部RAM中，从而可使有限的内部数据存储器得到充分利用。模拟再入栈及其再入栈指针可以通过配置文件“STARTUP.A51”进行调整，使用再入函数时应根据需要对配置文件进行适当修改。

3.5 C51编译器的数据调用协议

3.5.1 数据在内存中的存储格式

bit类型数据只有一位长度，不允许定义位指针和位数组。bit对象始终位于8051单片机片内可位寻址数据存储器空间（20~2FH），只要有可能，BL51连接定位器将对位对象进行覆盖操作。

char类型数据的长度为一个字节（8位），可存放在8051单片机片内或片外数据存储器。

int和short类型数据的长度为两个字节（16位），可存放在8051单片机片内或片外数据存储器。数据存储时按高字节地址在前、低字节地址在后的顺序存放，如个值为0x1234的int类型数据，在内存中的存储格式为

地址	+0	+1
内容	0x12	0x34

long类型数据的长度为4个字节（32位），可存放在8051单片机内部或外部数据存储器。数据存储时按高字节地址在前、低字节地址在后的顺序存放，如一个值为0x12345678的

long类型数据，在内存中的存储格式为

地址	+0	+1	+2	+3
内容	0x12	0x34	0x56	0x78

float类型数据的长度为4个字节（32位），可存放在8051单片机内部或外部数据存储器。一个float类型数据的数值范围是 $(-1)^S \times 2^{E-127} \times (1.M)$ ，在内存中按IEEE-754标准单精度32位浮点数的格式存储，即

地址	+0	+1	+2	+3
内容	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM

其中，S为符号位，0表示正，1表示负。E为用原码表示的阶码，占用8位二进制数，存放在两个字节中，E的取值范围为1~254。注意，实际上以2为底的指数要用E的值减去偏移量127，从而实际幂指数的取值范围为-126~+127。M为尾数的小数部分，用23位二进制数表示，存放在三个字节中。尾数的整数部分永远为1，因此不予保存，但它是隐含存在的。小数点位于隐含的整数位“1”的后面。

例如，一个值为-12.5的float类型数据，在内存中的存储格式为

地址	+0	+1	+2	+3
二进制内容	11000001	01001000	00000000	00000000
十六进制内容	0xC1	0x48	0x00	0x00

按上述规则很容易将用十六进制表示的数据0xC1480000转换为浮点数-12.5。

一个浮点数的正常数值范围为 $(-1)^S \times 2^{E-127} \times (1.M)$ 。其中， $E=0 \sim 255$ ， $S=\pm 1$ 。超过最大正常数值的浮点数就认为是无穷大，其阶码E为全1（即255），小数部分M为全0，表示为

$$\pm\infty = (-1)^S \times 2^{128} \times (1.000\dots 000) = \pm 2^{128}。$$

对于阶码E为全0，小数部分M也为全0的浮点数认为是0，表示为

$$(-1)^S \times 2^{-127} \times (1.000\dots 000) = \pm 2^{-127}。$$

绝对值最小的正常浮点数为阶码E为1，小数部分M为全0的数，表示为

$$(-1)^S \times 2^{-126} \times (1.000\dots 000) = \pm 2^{-126}。$$

除了正常数之外，界于 $+2^{-126} \sim +2^{-127}$ 和 $-2^{-126} \sim -2^{-127}$ 之间的数为非正常数。按IEEE754标准，浮点数的数值如果在正常数值之外，即为溢出错误，则用下面的二进制数表示，即

非正常数：NaN=0FFFFFFFH

正无穷：+INF=7F800000H

负无穷：-INF=FF800000H

C51编译器支持“基于存储器”的指针和“一般”指针。基于存储器类型data、idata和pdata的指针具有1个字节的长度，基于存储器类型xdata和code的指针具有2个字节的长度，一般指针具有3个字节的长度。在一般指针的3个字节中，第一个字节表示存储器类型，第二、第三个字节表示指针的地址偏移量，一般指针在内存中的存储格式为

地址	+0	+1	+2
内容	存储器类型	高字节地址偏移量	低字节地址偏移量

第一个字节中存储器类型的编码为

存储器类型	idata/data/bdata	xdata	pdata	code
编码值(8051)	0x00	0x01	0xFE	0xFF
编码值(8051Mx)	0x7F	0x00	0x00	0x80

采用一般指针时必须使用规定的存储器类型编码值，如果使用其他类型的值，将导致不可预测的后果。

例如，将xdata类型的地址0x1234作为一般指针表示为

地址	+0	+1	+2
内容	0x01	0x12	0x34

3.5.2 目标代码的段管理

段是程序代码或数据对象的存储器单位。程序代码被放入代码段。数据对象被放入数据段。段又分为绝对段和再定位段。绝对段只能在汇编语言程序中指定，包括代码和数据的绝对地址说明。绝对段在用连接定位器BL51连接时，已经分配的地址将不发生任何改变。再定位段是由C51编译器对C51源程序编译时所产生的，再定位段中代码或数据的存储器地址是浮动的，实际地址要由连接定位器BL51对程序模块进行连接时决定。再定位段可以保证在进行多模块程序连接时不会发生地址重叠现象，因此绝对段只是用于某些特殊场合，如访问某个固定的存储器I/O地址，或是提供某个中断向量的入口地址，而用C51编译器对C51源程序进行编译所产生的段都是再定位段。每一个再定位段都具有段名和存储器类型，绝对段则没有段名。下面介绍C51编译器对再定位段的管理方法。

为了适应不同要求和便于段管理，C51编译器在对C51源程序进行编译时，将程序中每个数据对象都转换成大写形式保存，并放入到相应的段中。C51编译器按表3.7规则将源程序中的函数名转换成目标文件中的符号名。BL51在连接定位时将使用目标文件符号名。

表3.7 C51编译器的函数名转换规则

函数声明	转换目标文件中的符号名	说 明
void func (void) ...	FUNC	无参数传递或不含寄存器参数的函数名不做改变地转入目标文件中，函数名只简单地转换成大写形式
void func (char) ...	_FUNC	带寄存器参数的函数名前面加上“_”前缀，表示这类函数包含有寄存器内的参数传递
void func (void) reentrant...	__FUNC	再入函数在函数名前面加上“__”前缀，表示该函数包含栈内的参数传递

完成函数名转换之后，Cx51编译器按以下规则将不同的数据对象组合到不同的数据段中。

(1) 全局变量

对于全局变量C51编译器按表3.8规则为每个模块生成各自的段名，具有相同存储器类型的全局变量被组合到同一个数据段中。每个明确定义了存储器类型的全局变量都有一个单独的数据段。段名由两个问号中间加一个存储器类型符号及紧接着的模块名(modulname)组成。模块名是不带路径和扩展名的源文件名。常数和字符串被放入一个独立的段中。各段名

表示在对应类型存储器空间的起始地址。

表3.8 C51编译器对全局变量的段名生成规则

段 名	存储器类型	说 明
?CO?modulname	Code	可执行程序存储器中的常数段
?XD?modulname	xdata	xdata型数据段（RAM空间）
?DT?modulname	data	data型数据段
?ID?modulname	idata	idata型数据段
?BI?modulname	bit	bit型数据段
?BA?modulname	bdata	bdata型数据段
?PD?modulname	pdata	pdata型数据段
?XC?modulname	const xdata	xdata型数据段（const ROM空间），需要用“OMF2”编译控制命令
?FC?modulname	const far	far型常数段（const ROM空间），需要用“OMF2”编译控制命令
?FD?modulname	far	far型数据段（RAM空间），需要用“OMF2”编译控制命令

（2）函数和局部全局变量

C51编译器为各个模块中的每个函数生成一个以?PR? function_name? modulname为名的代码（CODE）段。例如，如果程序模块SAMPLE.C中包含有一个名为ERROR_CHECK的函数，则代码（CODE）段的名为?PR? ERROR_CHECK? SAMPLE。

如果函数中包含有非寄存器传递的参数和无明确存储器类型声明的局部变量，则C51编译器除了生成该函数的代码段之外，还将生成一个字节类型的局部数据段（简称局部数据段）和一个位类型的局部数据段（简称局部位段）。局部位段用于存放在函数内部定义的可再定位的位类型变量和参数。局部数据段则用于存放除位类型以外的所有其他无明确存储器类型声明的局部变量和参数。对于已明确声明了存储器类型的函数局部变量，C51编译器根据变量的存储器类型将其组合到与本模块对应的全局数据段中，但这些变量仍属于定义它们的函数中的局部变量。函数的局部段（包含函数代码段、局部数据段和局部位段）的命名规则与函数的存储器模式有关，见表3.9。

表3.9 C51编译器的局部段命名规则

存储器模式	局部段类型	段 描 述	段 名
SMALL	code	函数代码	?PR? function_name? modul_name
	data	局部数据	?DT? function_name? modul_name
	bit	局部位段	?BI? function_name? modul_name
COMPAC	code	函数代码	?PR? function_name? modul_name
	pdata	局部数据	?PD? function_name? modul_name
	bit	局部位段	?BI? function_name? modul_name
LARGE	code	函数代码	?PR? function_name? modul_name
	xdata	局部数据	?XD? function_name? modul_name
	bit	局部位段	?BI? function_name? modul_name

C51编译器为局部数据段和局部位段建立一个可覆盖标志OVERLAYABLE，以便让连接定位器BL51在对目标程序进行连接定位时用于覆盖分析。

C51编译器允许通过寄存器传递最多3个参数，其他参数则需要通过固定的存储器区进行传递。对于通过固定存储器区进行参数传递的数据，按以下规则生成一个局部段：

局部数据段	?function_name? BYTE
局部位段	?function_name? BIT

段名都表示该段的起始地址，若函数func1需要通过固定的存储器区传递参数，则bit型参数将从地址?FUNC1?BIT开始传递，其他参数则从地址?FUNC1?BYTE开始传递。局部段名是全局共享的，因此它们的起始地址可被其他模块访问，从而为汇编语言程序调用C51函数提供了可能。

以上介绍的是C51编译器在对用户的C51源程序进行编译时，为实现多模块程序浮动连接而采用的段名管理方法。这些段名都包括在由连接定位器BL51所产生的MAP文件中，用户可以查看，以分析自己编写的C51源程序是否合理。

3.6 与汇编语言程序的接口

3.6.1 参数传递规则

C51编译器能对C51源程序进行高效率地编译，生成高效简洁形式的代码，在绝大多数场合采用C语言编程即可完成预期的任务。尽管如此，有时仍需要采用一定的汇编语言编程，如对于某些特殊I/O接口地址的处理、中断向量地址的安排、提高程序代码的执行速度等。为此，C51编译器提供了与汇编语言程序的接口规则，按此规则可以很方便地实现C51程序与汇编语言程序的相互调用。实际上，C51程序与汇编语言程序的相互调用也可视为函数的调用，只是此时函数是采用不同语言编写的而已。

C51程序函数和汇编语言函数在相互调用时，可利用 8051单片机的工作寄存器最多传递3个参数，见表3.10。

表3.10 参数传递的工作寄存器选择

传递的参数类型 传递的参数顺序	char或 单字节指针	int或2字节指针	long或float	一般指针
第一个参数	R7	R6（高字节），R7（低字节）	R4~R7	R3（存储类型）， R2（高字节），R1（低字节）
第二个参数	R5	R4（高字节），R5（低字节）	R4~R7	R3（存储类型）， R2（高字节），R1（低字节）
第三个参数	R3	R2（高字节），R3（低字节）	无	R3（存储类型）， R2（高字节），R1（低字节）

如果在调用时参数无寄存器可用，或是采用了编译控制命令NOREGPARMs，则可通过固定的存储器区域来传递参数。该存储器区域被称为参数传递段，其地址空间取决于编译时所选择的存储器模式，如

func1(int a);	//a是第一个int型参数，在R6，R7中传递。
func2(int b,int c,int *d);	//b在R6，R7中传递，c在R4，R5中传递，

```

func3(long e, long f);          /*d在R1, R2, R3中传递。
                                //e在R4, R5, R6, R7中传递,
                                //f 只能在参数传递段中传递。

func4(float g, char h);        //g在R4,R5,R6,R7中传递,
                                //h只能在参数传递段中传递。

```

当C51程序与汇编语言程序需要相互调用，并且参数的传递发生在参数传递段时，如果传递的参数是char、int、long和float类型的数据，则参数传递段的首地址由?functionname? BYTE的公共符号（PUBLIC）确定。如果传递的参数是bit类型的数据，则参数传递段的首地址由?functionname? BIT的公共符号（PUBLIC）确定。所有被传递的参数按顺序存放在以首地址开始递增的存储器区域内。参数传递段的存储器空间取决于所采用的编译模式，在SMALL模式下，参数传递段位于片内RAM空间，在COMPACT和LARGE模式下参数传递段位于外部RAM空间。

函数返回值被放入8051单片机寄存器内，返回值所占用工作寄存器见表3.11。

表3.11 函数返回值所占用工作寄存器

返回值类型	寄存器	说明
Bit	进位CY	返回值在进位标志CY中
(unsigned) char	R7	返回值在寄存器R7中
(unsigned) int	R6, R7	返回值高位在R6中，低位在R7中
(unsigned) long	R4~R7	返回值高位在R4中，低位在R7中
Float	R4~R7	32位IEEE格式，指数和符号位在R7中
一般指针	R3, R2, R1	R3放存储器类型，高位在R2中，低位在R1中

在汇编语言子程序中，当前选择的工作寄存器组及特殊功能寄存器ACC、B、DPTR和PSW的值都可能改变，当从汇编语言程序调用C语言函数时，必须无条件地假定这些寄存器的内容已被破坏。

如果在连接定位时采用了覆盖过程，则每个汇编语言子程序都将包含一个单独的程序段。这一点是必要的，因为在BL51连接定位器的覆盖分析中，函数之间的相互参考是通过子程序各自的段基准进行计算的。如果注意下面两点，则汇编语言子程序的数据区也可以包含在覆盖分析中：

① 所有段名都必须以C51编译器所规定的方法来建立；

② 每个具有局部变量的汇编语言函数都必须指定自己的局部数据段，这个局部数据段可以用来为其他函数访问作为参数传递，并且参数的传递要按顺序进行。

在汇编语言函数程序中，对于char、int、long和float类型的数据，局部数据段应以PUBLIC符号?_functionname? BYTE作为首地址，并在数据段中先按被传递参数的顺序定义若干字节，再定义其他局部变量数据字节。例如，在Small编译模式下，该数据段应按如下方式建立，即

```

RSEG ?DT?functionname? modulname ; 定义局部数据段名
?_functionname? BYTE:             ; 定义数据段首地址
    charVAL    DS    1             ; 按参数的传递顺序定义字节
    intVAL     DS    2

```



```
longVAL    DS    4
...                ; 定义其他局部变量字节
```

对于bit类型的数据，局部数据段应以PUBLIC符号?_functionname? BIT作为首地址，并按被传递参数的顺序先定义若干位，再定义其他局部变量位，如

```
RSEG ?BI?functionname? modulname ; 定义局部数据段名
?_functionname? BIT:              ; 定义数据段首地址
    bitVAL1    DBIT    1          ; 按参数的传递顺序定义位
    bitVAL2    DBIT    1
    ...                ; 定义其他局部变量位
```

这样定义的局部数据段可为其他函数访问作为参数传递，所有参数都将按顺序逐个传递。
下面是一个在Small编译模式下，C语言函数调用汇编语言函数的例子，可以清楚地了解参数的传递过程。

例3-1 C语言函数调用汇编语言函数的参数传递过程。
C51源程序C_CALL.C文件清单。

```
#pragma code small          //指定编译模式
extern int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e); //说明被调函数
void C_call() {              //主调函数
    int v_a;char v_b;bit v_c;long v_d;bit v_e;                //局部变量
    int A_ret;
    A_ret=afunc(v_a, v_b, v_c, v_d, v_e);                      //函数调用
}
```

文件中定义了两个函数：主调函数void C_call()和被用的外部函数extern int afunc (int v_a,char v_b,bit v_c,long v_d,bit v_e)。被调函数在另一个模块文件AFUNC.A51中采用汇编语言编写，函数中有5个参数，函数调用时最多可利用8051单片机的工作寄存器传递3个参数，因此只有参数v_a在寄存器R6、R7中传递（高位在R6，低位在R7），参数v_b在R5中传递。而参数v_c、v_d和v_e将在参数传递段中传递。程序编译时，采用了Small模式，参数传递段将位于8051单片机片内数据存储器DATA区。另外，函数afunc()是int类型的函数，所以函数afunc()的返回值在工作寄存器R6（高位）、R7（低位）中。

在被调用的汇编语言程序函数afunc()中，将C51程序函数C_call()传递过来的5个参数int v_a、char v_b、bit v_c、long v_d和bit v_e，分别放入局部变量a、b、c、d和e中，因此该汇编语言程序函数是包含有局部变量的。由于C51程序模块指定了Small编译模式，因此参数传递将在内部数据存储器DATA区域进行。汇编语言程序函数必须按C51编译器关于Small模式下段名规则建立相应的局部数据段，即对于需要利用工作寄存器进行参数传递的函数，函数名afunc前面要加一个下画线，还需要给出正确的参数传递段地址。

汇编语言程序AFUNC.A51文件清单。

NAME	AFUNC	
?PR?_afunc?AFUNC	SEGMENT CODE	; 定义程序代码段
?DT?_afunc?AFUNC	SEGMENT DATA OVERLAYABLE	; 定义可覆盖局部数据段
?BI?_afunc?AFUNC	SEGMENT BIT OVERLAYABLE	; 定义可覆盖局部位段
	PUBLIC ?_afunc?BIT	; 公共符号定义

```

PUBLIC ?_afunc?BYTE
PUBLIC _afunc
RSEG ?DT?_afunc?AFUNC ;可覆盖局部数据段
?_afunc?BYTE: ;起始地址
    v_a?040: DS 2 ;定义传递参数字节
    v_b?041: DS 1
    v_d?043: DS 4
    ORG 7
    a?045: DS 2 ;定义其他局部变量
    b?046: DS 1
    d?048: DS 4
    retval?050: DS 2 ;返回值
RSEG ?BI?_afunc?AFUNC ;可覆盖局部位段
?_afunc?BIT: ;起始地址
    v_c?042: DBIT 1 ;定义传递数据位
    v_e?044: DBIT 1
    ORG 2
    c?047: DBIT 1 ;定义其他局部变量位
    e?049: DBIT 1
RSEG ?PR?_afunc?AFUNC ;程序代码段
_afunc: ;起始地址
    USING 0
    MOV a?045,R6 ;a=v_a
    MOV a?045+01H,R7
    MOV b?046,R5 ;b=v_b
    MOV C,v_c?04 ;c=v_c2
    MOV c?047,C ;d=v_d
    MOV d?048+03H,v_d?043+03H
    MOV d?048+02H,v_d?043+02H
    MOV d?048+01H,v_d?043+01H
    MOV d?048,v_d?043
    MOV C,v_e?044 ;e=v_e
    MOV e?049,C
    MOV R6,retval?050 ;函数返回值高位
    MOV R7,retval?050+01H ;函数返回值低位
    RET
END

```

C51编译器提供了一个十分有用的编译控制命令SRC。编写汇编语言函数之前先用C51编写相应函数，对该函数单独采用SRC命令进行编译，产生一个扩展名为SRC的汇编语言源文件，再对该SRC文件按需要进行必要的修改，可以很方便地写出汇编语言函数。用这种方法有两大优点：一是用C51编写函数，能有效地提高开发效率；二是基本不用考虑C51函数名的转换、段的命名、参数传递等规则，而是直接在SRC文件上进行汇编语言代码修改，琐碎的工作都交由编译器完成，非常方便。

C51源程序AFUNC.C文件清单。

```
#pragma src (AFUNC.A51) small
```

```
int afunc(int v_a,char v_b,bit v_c,long v_d,bit v_e) {
    int a;char b;bit c;long d;bit e;
    int retval;
    a=v_a;b=v_b;c=v_c;d=v_d;e=v_e;
    return(retval);
}
```

对C51源程序AFUNC.C文件用SRC命令进行编译后,即可生成上面的汇编语言程序文件AFUNC.A51。需要注意的是,对模块文件AFUNC.C编译时,必须采用与模块文件C_CALL.C相同的编译模式,当调用有参函数时这一点十分重要。如果两个文件采用不同的编译模式,那么它们将采用不同的存储器区域作为参数传递段空间,这将导致不能正确地进行参数传递。

3.6.2 C51与汇编语言混合编程举例

采用C51与汇编语言混合编程,程序的主体部分用C语言编写,对执行时间具体硬件操作要求严格的部分用汇编语言编写。这种方法可以将C语言和汇编语言的优点结合起来。下面介绍两种最常用的混合编程方法。

例3-2 C51程序调用汇编语言函数。

Proteus仿真电路如图3.1所示,采用C51程序调用汇编语言函数的方式实现P1端口驱动数码管显示。

C51程序文件如下:

```
#include<reg52.h>
extern void delay(unsigned char t);    //说明被调的汇编语言函数

main() {
    while(1) {
        P1=0x00;        //从P1口点亮数码管
        delay(0x10);     //短延时
        P1=0xff;        //从P1口熄灭数码管
        delay(0xff);     //长延时
    }
}
```

汇编语言函数文件如下:

```
NAME            DELAY
?PR?_delay?DELAY SEGMENT CODE    ;定义程序代码段
PUBLIC _delay
RSEG ?PR?_delay?DELAY
_delay: MOV A,R7                ;延时参数通过R7传递
        MOV R6,A
LP3:    MOV R5,#0FFH
LP4:    DJNZ R5,LP4
        DJNZ R6,LP3
        RET
END
```

新建一个 μ Vision3项目，分别将C语言程序文件和汇编语言函数文件加入到项目之中，如图3.2所示，最后对整个项目进行编译连接，生成可执行的目标代码。

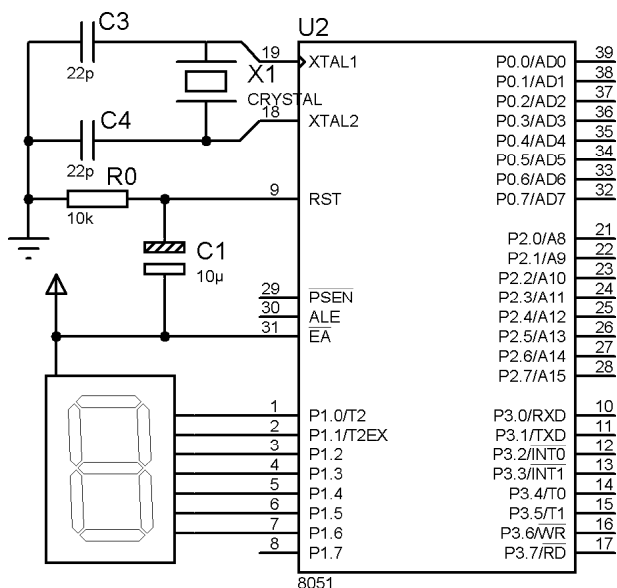


图3.1 Proteus仿真电路

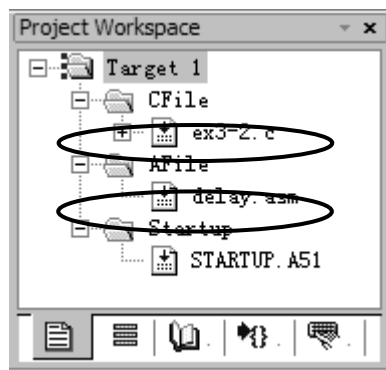


图3.2 混合编程项目窗口

将生成的可执行目标代码文件装入如图3.1所示8051单片机中，可以看到数码管不断闪烁显示。

例3-3 在C51程序中嵌入汇编语言指令。

将上例程序改写为如下C51源程序文件，通过预编译指令“#pragma asm”和“#pragma asm”在C51程序中插入汇编语言指令。

```
#include<reg52.h>
main(){
    while(1){
        P1=0x00;          // C51语句
#pragma asm              // 插入汇编指令
        MOV R6,#080H      // 短延时
LP3:    MOV R5,#0FFH
LP4:    DJNZ R5,LP4
        DJNZ R6,LP3
#pragma endasm           // 结束汇编
        P1=0xff;          // C51语句
#pragma asm              // 插入汇编指令
        MOV R6,#0FFH      // 长延时
LP5:    MOV R5,#0FFH
LP6:    DJNZ R5,LP6
        DJNZ R6,LP5
#pragma endasm           // 结束汇编
    }
}
```

新建一个 μ Vision3项目，将改写后的例3-3源程序文件加入到项目中。在项目窗口将鼠标指向包含汇编代码的C语言文件并单击右键，选择右键菜单的“Options for File ex3-3.c”选项，弹出如图3.3所示窗口，单击选中窗口右边的复选框“Generate Assembler SRC File”和“Assemble SRC File”，使其由灰色变成黑色状态。

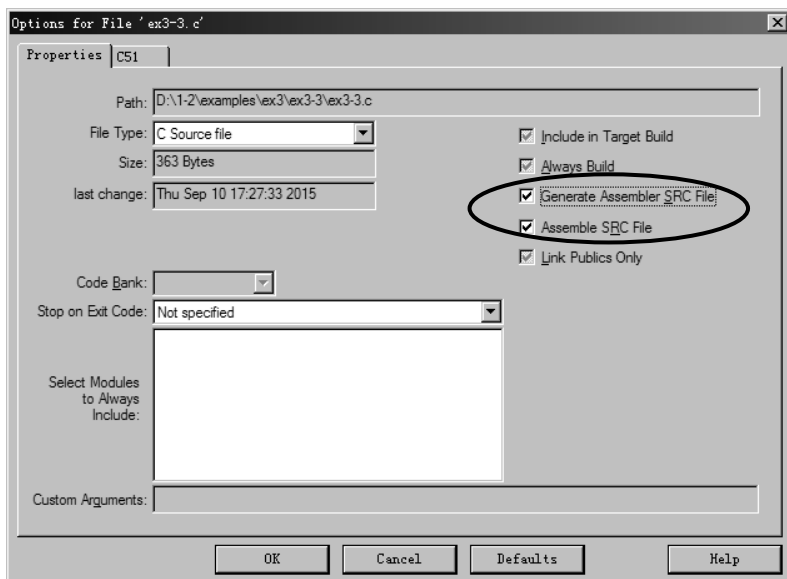


图3.3 Options for File窗口

注意，如果没有这一步，编译时将会出现错误“'asm/endasm' requires src-control to be active”而无法通过编译。

根据所选择的编译模式，把相应的库文件（Small 模式下为C51S.Lib，Compact模式下为C51C.Lib，Large 模式下为C51L.Lib）加入到当前项目中，如图3.4所示。

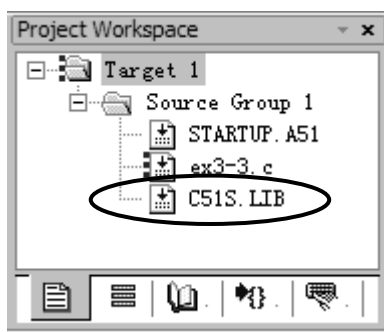


图3.4 通过项目窗口中加入库文件

注意，如果没有这一步，则编译时会出现错误“UNRESOLVED EXTERNAL SYMBOL”而无法通过编译。

编译连接，生成可执行目标代码。将生成的可执行目标代码文件装入如图3.1所示8051单片机中，可以看到数码管的显示结果与例3-2完全一样。

3.7 绝对地址访问

在进行8051单片机应用系统程序设计时，用户十分关心如何直接操作系统的各个存储器地址空间。C51程序经过编译之后产生的目标代码具有浮动地址，其绝对地址必须经过BL51连接定位后才能确定。为了能够在C51程序中直接对任意指定的存储器地址进行操作，可以采用扩展关键字“_at_”、指针、预定义宏及连接定位控制命令，分别介绍如下。

3.7.1 采用扩展关键字“_at_”或指针定义变量的绝对地址

在C语言源程序中定义变量时，可以利用C51编译器提供的扩展关键字“_at_”来对指定变量的存储器空间绝对地址，一般格式为

[存储器类型] 数据类型 标识符 _at_ 地址常数

其中，“存储器类型”为idata、data、xdata等C51编译器能够识别的所有类型，如果省略该选项，则按编译模式SMALL、COMPACT或LARGE规定的默认存储器类型确定变量的存储空间；“数据类型”除了可用int、long、float等基本类型外，还可以采用数组、结构等复杂数据类型；标识符为要定义的变量名；地址常数规定了变量的绝对地址，它必须位于有效存储器空间之内。下面是几个采用关键字“at”进行变量的绝对地址定位的例子。

```
struct link {
    struct link idata *next;
    char code *test;
};
idata struct link list _at_ 0x40; // 结构变量list位于idata空间地址0x40
xdata char text[256] _at_ 0xE000; // 数组array位于xdata空间地址0xE000
xdata int il _at_ 0x8000;          // int变量il位于xdata空间地址0x8000
```

利用扩展关键字“_at_”定义的变量称为“绝对变量”，对该变量的操作就是对指定存储器空间绝对地址的直接操作，因此不能对“绝对变量”进行初始化，对于函数和位（bit）类型变量不能采用这种方法进行绝对地址定位。采用关键字“_at_”所定义的绝对变量必须是全局变量，在函数内部不能采用“_at_”关键字指定局部变量的绝对地址。另外，在XDATA空间定义全局变量的绝对地址时，还可以在变量前面加一个关键字“volatile”，这样对该变量的访问就不会被C51编译器优化掉。

利用基于存储器的指针也可以指定变量的存储器绝对地址，其方法是先定义一个基于存储器的指针变量，然后对该变量赋以存储器绝对地址值。下面是几个利用基于存储器的指针进行变量的绝对地址定位的例子。

```
char xdata temp _at_ 0x4000; /* 定义全局变量temp，地址为XDATA空间0x4000 */
void main( void ) {
    char xdata *xdp;          // 定义一个指向XDATA存储器空间的指针
    char data *dp;            // 定义一个指向DATA存储器空间的指针
    xdp = 0x2000;             // XDATA指针赋值，指向XDATA存储器地址0002h
```

```

temp = *xdp;           // 读取XDATA空间地址0x2000的内容送往0x4000单元
*xdp = 0xAA;           // 将数据0xAA送往XDATA空间0x2000地址单元
dp = 0x30;             // DATA指针赋值, 指向DATA存储器地址30H
*dp = 0xBB;            // 将数据0xBB送往指定的DATA空间地址
}

```

3.7.2 采用预定义宏指定变量的绝对地址

Cx51编译器的运行库中提供了如下一套预定义宏:

CBYTE	CWORD	FARRAY
DBYTE	DWORD	FCARRAY
PBYTE	PWORD	FCVAR
XBYTE	XWORD	FVAR

这些宏定义包含在头文件“ABSACC.H”中, 在C语言源程序中可以利用这些宏来指定变量的绝对地址, 如

```

#include <ABSACC.H>
char c_var;
int i_var;
XBYTE[0x12]=c_var;      // 向XDATA存储器地址0012H写入数据c_var
i_var=XWORD[0x100];     // 从XDATA存储器地址0200H中读取数据并赋值给i_var

```

上面第二条赋值语句中采用的是XWORD[0x100], 它是对地址“2*0x100”进行操作。该语句的意义是将字节地址0x200和0x201的内容取出来并赋值给int型变量i_var。注意, 不要将XWORD与XBYTE混淆。如果将这条语句改成

```
i_var=XWORD[0x100/2];
```

则读取的就是0x100和0x101地址单元中的内容了。用户可以充分利用C51运行库中提供的预定义宏来进行绝对地址的直接操作。例如, 可以采用如下方法定义一个D/A转换接口地址, 每向该地址写入一个数据即可完成一次D/A转换:

```

#include <ABSACC.H>
#define DAC0832 XBYTE[0x7fff] // 定义DAC0832端口地址
DAC0832=0x80;                // 启动一次D/A转换

```

➡ 3.8 Keil C51库函数

丰富的可直接调用库函数是Keil C51的一个重要特征, 正确而灵活地使用库函数可使程序代码简单, 结构清晰, 易于调试和维护。每个库函数都在相应头文件中给出函数原型声明, 用户如果需要使用库函数, 则必须在源程序的开始处采用预处理器命令#include将有关的头文件包含进来。如果省略了头文件, 将不能保证函数的正确运行。下面简要介绍Keil C51编译器提供的库函数。

3.8.1 本征库函数

本征库函数是指编译时直接将固定的代码插入到当前行，而不是用汇编语言中的“ACALL”和“LCALL”指令来实现调用，从而大大提高函数的访问效率。非本征库函数则必须由“ACALL”和“LCALL”指令来实现调用。Keil C51的本征库函数有9个，数量虽少，但非常有用，见表3.12。使用本征函数时，C51源程序中必须包含预处理命令#include <intrins.h>。

表3.12 本征库函数

函数名及定义	功 能 说 明
unsigned char _crol_(unsigned char val, unsigned char n);	将字符型数据val循环左移n位，相当于RL指令
unsigned int _lrol_(unsigned int val, unsigned char n);	将整型数据val循环左移n位，相当于RL指令
unsigned long _lrol_(unsigned long val, unsigned char n);	将长整型数据val循环左移n位，相当于RL指令
unsigned char _cror_(unsigned char val, unsigned char n);	将字符型数据val循环右移n位，相当于RR指令
unsigned int _iror_(unsigned int val, unsigned char n);	将整型数据val循环右移n位，相当于RR指令
unsigned long _lror_(unsigned long val, unsigned char n);	将长整型数据val循环右移n位，相当于RR指令
bit _testbit_(bit x);	相当于JBC bit 指令
unsigned char _chkfloat_(float ual);	测试并返回浮点数状态
void _nop_(void);	产生一个NOP指令

3.8.2 字符判断转换库函数

字符判断转换库函数的原型声明在头文件CTYPE.H中定义。表3.13列出了字符判断转换库函数的功能说明。

表3.13 字符判断转换库函数的功能说明

函数名及定义	功 能 说 明
bit isalpha(char c);	检查参数字符是否为英文字母，是则返回1，否则返回0
bit isalnum(char c);	检查参数字符是否为英文字母或数字字符，是则返回1，否则返回0
bit iscntrl(char c);	检查参数值是否为控制字符（值在0x00~0x1f之间或等于0x7f），如果是则返回1，否则返回0
bit isdigit(char c);	检查参数的值是否为十进制数字0~9，是则返回1，否则返回0
bit isgraph(char c);	检查参数是否为可打印字符（不包括空格），可打印字符的值域为0x21~0x7e，是则返回1，否则返回0
bit isprint(char c);	除了与isgraph相同之外，还接受空格符（0x20）
bit ispunct(char c);	检查字符参数是否为标点、空格或格式字符。如果是空格或是32个标点和格式字符之一（假定使用ASCII字符集中128个标准字符），则返回1，否则返回0
bit islower(char c);	检查参数字符的值是否为小写英文字母，是则返回1，否则返回0
bit isupper(char c);	检查参数字符的值是否为大写英文字母，是则返回1，否则返回0
bit isspace(char c);	检查参数字符是否为下列之一：空格、制表符、回车、换行、垂直制表符和送纸（值为0x09~0x0d，或为0x20），是则返回1，否则返回0
bit isxdigit(char c);	检查参数字符是否为16进制数字字符，是则返回1，否则返回0

(续表)

函数名及定义	功 能 说 明
char toint(char c);	将ASCII字符的0~9、a~f（大小写无关）转换为16进制数字，对于ASCII字符的0~9，返回值为0H~9H，对于ASCII字符的a~f（大小写无关），返回值为0AH~0FH
char tolower(char c);	将大写字符转换成小写形式，如果字符参数不在'A'~'Z'之间，则该函数不起作用
char _tolower(char c);	将字符参数c与常数0x20逐位相或，从而将大写字符转换为小写字符
char toupper(char c);	将小写字符转换为大写形式，如果字符参数不在'a'~'z'之间，则该函数不起作用
char _toupper(char c);	将字符参数C与常数0XDF逐位相与，从而将小写字符转换为大写字符
char toascii(char c);	该宏将任何字符型参数值缩小到有效的ASCII范围之内，即将参数值和0x7F相与从而去掉第7位以上的所有数位

3.8.3 输入、输出库函数

输入、输出库函数的原型声明在头文件STDIO.H中定义，通过8051系列单片机的串行口工作，如果希望支持其他I/O接口，则只需要改动_getkey()和putchar()函数，库中所有其他I/O支持函数都依赖于这两个函数模块，在使用8051系列单片机的串行口之前，应先对其进行初始化。例如，以2400波特率（12MHz时钟频率）初始化串行口的语句为

```
SCON=0x52;          /* SCON 置初值 */
TMOD=0x20;          /* TMOD 置初值 */
TH1=0xf3;           /* T1 置初值 */
TR1=1;              /* 启动 T1 */
```

表3.14列出了输入、输出库函数的功能说明。

表3.14 输入、输出库函数的功能说明

函数名及定义	功 能 说 明
char _getkey(void);	等待从8051串口读入一个字符并返回读入的字符，这个函数是改变整个输入端口机制时应做修改的唯一一个函数
char getchar(void);	使用_getkey从串口读入字符，并将读入的字符马上传给putchar 函数输出，其他与_getkey函数相同
char * gets(char *s, int n);	该函数通过getchar从串口读入一个长度为n的字符串并存入由's'指向的数组。输入时，一旦检测到换行符就结束字符输入。输入成功时，返回传入的参数指针，失败时返回NULL
char ungetchar(char c);	将输入字符回送输入缓冲区，因此下次gets或getchar可用该字符。成功时返回char型值c，失败时返回EOF，不能用ungetchar处理多个字符
char putchar(char c);	通过8051串行口输出字符，与函数_getkey一样，这是改变整个输出机制所需修改的唯一一个函数
int printf(const char * fmstr [,argument]...);	以第一个参数指向字符串制定的格式通过8051串行口输出数值和字符串，返回值为实际输出的字符数
int sprintf(char * s, const char * fmstr [,argument] ...);	与printf的功能相似，但数据不是输出到串行口，而是通过一个指针s送入内存缓冲区，并以ASCII码的形式储存。参数fmstr与函数printf一致
int puts(const char * s);	利用putchar函数将字符串和换行符写入串行口，错误时返回EOF，否则返回0

(续表)

函数名及定义	功 能 说 明
<code>int scanf(const char * fmstr [,argument] ...);</code>	在格式控制串的控制下, 利用 <code>getchar</code> 函数从串行口读入数据, 每遇到一个符合格式控制串 <code>fmstr</code> 规定的值, 就将它按顺序存入由参数指针 <code>argument</code> 指向的存储单元。注意, 每个参数都必须是指针。 <code>scanf</code> 返回它所发现并转换的输入项数, 若遇到错误, 则返回EOF
<code>int sscanf(char * s, const char * fmstr [,argument] ...);</code>	与 <code>scanf</code> 的输入方式相似, 但字符串的输入不是通过串行口, 而是通过指针 <code>s</code> 指向的数据缓冲区
<code>void vprintf(const char * s, char * fmstr, char * argptr);</code>	将格式化字符串和数据值以ASCII码的形式输出到串行口, 该函数类似于 <code>printf()</code>
<code>void vsprintf(const char * s, char * fmstr, char * argptr);</code>	将格式化字符串和数据值以ASCII码的形式输出到由指针 <code>s</code> 指向的内存缓冲区, 该函数类似于 <code>sprintf()</code>

最常用的输出库函数为`printf()`。该函数以一定的格式, 通过8051的串行口输出数值和字符串, 返回值为实际输出的字符数。

`printf()`函数的第一个参数`fmstr`是格式控制字符串。参数`argument`可以是字符串指针、字符或数值, 允许作为`printf`参数的总字节数受C51库限制, 由于8051系列单片机结构上存储空间有限, 在`small`和`compact`编译模式下最大可传递15个字节的参数(5个指针, 或1个指针和3个长字), 在`large`编译模式下, 最多可传递40个字节的参数。

格式控制字符串`fmstr`具有如下形式(方括号内是可选项):

```
%[flags][width][.precision][{b|B|l|L}]type
```

其中, 可选项`flag`被称为标志字符, 用于控制输出位置、符号、小数点及8进制和16进制数的前缀等。其内容和意义见表3.15。

表3.15 flag选项的内容和意义

flag选项	意 义
-	输出左对齐
+	输出如果是有符号数值, 则在前面加上+/-号
空格	输出值如果为正则左边补以空格, 否则不显示空格
#	如果它与0、x或X联用, 则在非0输出值前面加上0、0x或0X。当它与值类型字符g、G、f、e、E联用时, 使输出值中产生一个十进制的小数点
*	忽略指定格式

可选项`width`用来定义欲显示的字符数, 它必须是一个正的十进制数, 如果实际显示的字符数小于`width`, 则在输出左端补以空格, 如果`width`以0开始, 则在左端补以0。

可选项`precision`用来表示输出精度, 它是由小圆点“.”加上一个非负的十进制整数构成的。指定精度时可能会导致输出值被截断, 或在输出浮点数时引起输出值的四舍五入。可以用精度来控制输出字符的数目、整数值的位数或浮点数的有效位数。也就是说, 对于不同的输出格式, 精度具有不同的意义。

可选字符`b`或`B`和`l`或`L`通常与格式转换字符一起使用, 具体意义见表3.16。

表3.16 可选字符b、B、l、L的意义

b, B	当它们与格式类型字符d、o、u、x或X联用时，使参数类型被接受为[unsigned]char，如%bu、%bx等
l, L	它们与格式类型字符d、o、u、x或X联用，使参数类型被接受为[unsigned]long，如%ld、%lx等

type被称为输出格式转换字符。其内容和意义见表3.17。

表3.17 type选项的内容和意义

格式转换字符	类 型	输 出 格 式
d	int	有符号十进制数（16位）
u	unsigned int	无符号十进制数
o	unsigned int	无符号八进制数
x, X	unsigned int	无符号十六进制数
f	float	[-]dddd.dddd形式的浮点数
e,E	float	[-]d.ddddE[sign]dd形式的浮点数
g, G	float	选择e或f形式中更紧凑的一种输出格式
c	char	单个字符
s	一般指针	结束符为“\0”的字符串
p	一般指针	带存储器类型标志和偏移的指针M:aaaa。其中，M:=C(ode), D(ata), I(data), P(data) a=指针偏移值

例3-4 输出库函数应用。

```
#include <stdio.h>
void tst_printf (void) {
    char a;
    int b;
    long c;
    unsigned char x;
    unsigned int y;
    unsigned long z;
    float f,g;
    char buf [] = "Test String";
    char *p = buf;
    SCON=0x52;          /* 串行口初始化 */
    TMOD=0x20;
    TH1=0xf3;
    TR1=1;
    a = 1;   b = 12365;   c = 0x7FFFFFFF;
    x = 'A';   y = 54321;   z = 0x4A6F6E00;
    f = 10.0;   g = 22.95;
    printf ("char %bd int %d long %ld\n",a,b,c);
    printf ("Uchar %bu Uint %u Ulong %lu\n",x,y,z);
    printf ("xchar %bx xint %x xlong %lx\n",x,y,z);
    printf ("String %s is at address %p\n",buf,p);
    printf ("%f != %g\n", f, g);
    printf ("%*f != %*g\n", 8, f, 8, g);
}
```

最常用的输入库函数为scanf()。该函数以一定的格式，通过8051的串行口输入数值和字符串，返回它所发现并转换的输入项数，若遇到错误，则返回EOF。

scanf()函数的第一个参数fmstr是格式控制字符串，从串行口输入数据时，每遇到一个符合格式控制串fmstr规定的值，就将它按顺序存入由参数指针argument指向的存储单元。注意，每个参数都必须是指针。

格式控制串fmstr具有如下形式（方括号内为可选项）：

```
%[*][width][b|h|l]type
```

其中，可选项width是一个十进制的正整数，用来控制输入数据的最大宽度或字符数目。不超过规定宽度的字符从输入流中读出，并被转换到相应的变量，但是如果先遇到一个空格符或无法辨识的字符，则读入的字符数可能会小于宽度值。

可选字符b、h、l可以直接位于输入格式转换字符之前，其意义见表3.18。

表3.18 可选字符b、h、l的意义

b, h	它们用作格式类型d,o,u和x的前缀，用这个前缀可将参数定义为字符指针，指示输入整型数，如%bu、%bx
l	它被用作格式类型d,o,u和x的前缀，用这个前缀可将参数定义成长指针，指示输入长整数，如%lu，%lx

type被称为输入格式转换字符，其内容和意义见表3.19。

表3.19 type选项的内容和意义

格式转换字符	类 型	输 入 格 式
d	int *	有符号的十进制数
i	int *	有符号的十进制、十六进制、八进制数
u	unsigned int *	无符号的十进制数
o	unsigned int *	无符号的八进制数
x	unsigned int *	无符号的十六进制数
f,e,g	float *	浮点数
c	char *	一个字符
s	char *	一个字符串

例3-5 输入库函数应用

```
#include <stdio.h>
void tst_scanf (void) {
    char a;
    int b;
    long c;
    unsigned char x;
    unsigned int y;
    unsigned long z;
    float f,g;
    char d, buf [10];
    int argsread;
    SCON=0x52;      /* 串行口初始化 */
    TMOD=0x20;
```

```

TH1=0xf3;
TR1=1;
printf ("Enter a signed byte, int, and long\n");
argsread = scanf ("%bd %d %ld", &a, &b, &c);
printf ("%d arguments read\n", argsread);
printf ("Enter an unsigned byte, int, and long\n");
argsread = scanf ("%bu %u %lu", &x, &y, &z);
printf ("%d arguments read\n", argsread);
printf ("Enter a character and a string\n");
argsread = scanf ("%c %9s", &d, buf);
printf ("%d arguments read\n", argsread);
printf ("Enter two floating-point numbers\n");
argsread = scanf ("%f %f", &f, &g);
printf ("%d arguments read\n", argsread);
}

```

3.8.4 字符串处理库函数

字符串处理库函数的原型声明包含在头文件STRING.H中。字符串函数通常接收指针串作为输入值。一个字符串应包括两个或多个字符。字符串的结尾以空字符表示。在函数memcmp、memcpy、memchr、memccpy、memset和memmove中，字符串的长度由调用者明确规定，这些函数可工作在任何模式。表3.20列出了字符串处理库函数的功能说明。

表3.20 字符串处理库函数的功能说明

函数名及定义	功 能 说 明
void * memchr(void * s1, char val, int len);	顺序搜索字符串s1的前len个字符以找出字符val，成功时返回s1中指向val的指针，失败时返回NULL
char memcmp(void * s1, void * s2, int len);	逐个字符比较串s1和s2的前len个字符，成功（相等）时返回0，如果串s1大于或小于s2，则相应地返回一个正数或一个负数
void * memcpy(void * dest, void * src, int len);	从src所指向的内存中拷贝len个字符到dest中，返回指向dest中最后一个字符的指针。如果src与dest发生交迭，则结果是不可预测的
void * memccpy(void * dest, void * src, char val, int len);	拷贝src中len个元素到dest中。如果实际拷贝了len个字符，则返回NULL。 拷贝过程在拷贝完字符val后停止，此时返回指向dest中下一个元素的指针
void * memmove(void * dest, void * src, int len);	工作方式与memcpy相同，但拷贝的区域可以交迭
void memset(void * s, char val, int len);	用val来填充指针s中len个单元
void * strcat(char * s1, char * s2);	将串s2拷贝到s1的尾部。strcat假定s1所定义的地址区域足以接受两个串。返回指向s1串中第一个字符的指针
char * strncat(char * s1, char * s2, int n);	拷贝串s2中n个字符到s1的尾部，如果s2比n短，则只拷贝s2（包括串结束符）
char strcmp(char * s1, char * s2);	比较串s1和s2，如果相等，则返回0；如果s<s2，则返回一个负数；如果s1>s2，则返回一个正数
char strncmp(char * s1, char * s2, int n);	比较串s1和s2中的前n个字符。返回值与strcmp相同

(续表)

函数名及定义	功 能 说 明
<code>char * strcpy(char * s1, char * s2);</code>	将串s2, 包括结束符拷贝到s1中, 返回指向s1中第一个字符的指针
<code>char * strncpy(char * s1, char * s2, int n);</code>	与strcpy相似, 但它只拷贝n个字符。如果s2的长度小于n, 则s1串以0补齐到长度
<code>int strlen(char * s1);</code>	返回串s1中的字符个数, 不包括结尾的空字符
<code>char * strstr(const char * s1, char * s2);</code>	搜索字符串s2第一次出现在s1中的位置, 并返回一个指向第一次出现位置开始处的指针。如果字符串s1中不包括字符串s2, 则返回一个空指针
<code>char * strchr(char * s1, char c);</code>	搜索s1串中第一个出现的字符c, 如果成功, 则返回指向该字符的指针, 否则返回NULL。被搜索的字符可以是串结束符, 此时返回值是指向串结束符的指针
<code>int strpos(char * s1, char c);</code>	与strchr类似, 但返回的是字符c在串s1中第一次出现的位置值, 没有找到, 则返回-1, s1串首字符的位置值是0
<code>char * strrchr(char * s1, char c);</code>	搜索s1串中最后一个出现的字符c, 如果成功, 则返回指向该字符的指针, 否则返回NULL。被搜索的字符可以是串结束符
<code>int strrpos(char * s1, char c);</code>	与strrchr相似, 但返回值是字符c在s1串中最后一次出现的位置值, 没有找到, 则返回-1
<code>int strspn(char * s1, char * set);</code>	搜索s1串中第一个不包括在set串中的字符, 返回值是s1中包括在set里的字符个数。如果s1中所有字符都包括在set里面, 则返回s1的长度 (不包括结束符)。如果set是空串, 则返回0
<code>int strcspn(char * s1, char * set);</code>	与strspn相似, 但它搜索的是s1串中第一个包含在set里的字符
<code>char * strpbrk(char * s1, char * set);</code>	与strspn相似, 但返回指向搜索到字符的指针, 而不是个数, 如果未找到, 则返回NULL
<code>char * strpbrk(char * s1, char * set);</code>	与strpbrk相似, 但它返回s1中指向找到的set字符集中最后一个字符的指针

3.8.5 类型转换及内存分配库函数

类型转换及内存分配库函数的原型声明包含在头文件STDLIB.H中, 利用该库函数可以完成数据类型转换及存储器分配操作。表3.21列出了类型转换及内存分配库函数的功能说明。

表3.21 类型转换及内存分配库函数的功能说明

函数名及定义	功 能 说 明
<code>float atof(char * s1);</code>	将字符串s1转换成浮点数值并返回它, 输入串中必须包含与浮点值规定相符的数。该函数在遇到第一个不能构成数字的字符时, 停止对输入字符串的读操作
<code>long atoll(char * s1);</code>	将字符串s1转换成一个长整型数值并返回它, 输入串中必须包含与长整型格式相符的字符串。该函数在遇到第一个不能构成数字的字符时, 停止对输入字符串的读操作
<code>int atoi(char * s1);</code>	将串s1转换成整型数并返回它。输入串中必须包含与整型数格式相符的字符串。该函数在遇到第一个不能构成数字的字符时, 停止对输入字符串的读操作
<code>void * calloc(unsigned int n, unsigned int size);</code>	为n个元素的数组分配内存空间, 数组中每个元素的大小为size, 所分配的内存区域用0进行初始化。返回值为已分配的内存单元起始地址, 如不成功, 则返回0

(续表)

函数名及定义	功 能 说 明
void free(void xdata * p);	释放指针p所指向的存储器区域，如果p为NULL，则该函数无效，p必须是以前用calloc、malloc或realloc函数分配的存储器区域。调用free函数后，被释放的存储器区域就可以参加以后的分配
void init_mempool(void xdata * p, unsigned int size);	对可被函数calloc、free、malloc和realloc管理的存储器区域进行初始化，指针p表示存储区的首地址，size表示存储区的大小
void * malloc(unsigned int size);	在内存中分配一个size字节大小的存储器空间，返回值为一个size大小对象所分配的内存指针。如果返回NULL，则无足够的内存空间可用
void * realloc(void xdata * p, unsigned int size)	用于调整先前分配的存储器区域大小。参数p指示该存储区域的起始地址，参数size表示新分配存储器区域的大小。原存储器区域的内容被复制到新存储器区域中。如果新区域较大，多出的区域将不做初始化。realloc 返回指向新存储区的指针，如果返回NULL，则无足够大的内存可用，这时将保持原存储区不变
int rand();	返回一个0~32767之间的伪随机数，对rand的相继调用将产生相同序列的随机数
void srand(int n);	用来将随机数发生器初始化成一个已知（或期望）值
unsigned long strtod(const char * s, char **ptr);	将字符串s转换为一个浮点型数据并返回它，字符串前面的空格、/、tab符被忽略
long strtol(const char * s, char **ptr, unsigned char base);	将字符串s转换为一个long型数据并返回它，字符串前面的空格、/、tab符被忽略
long strtoul(const char * s, char **ptr, unsigned char base);	将字符串s转换为一个unsigned long型数据并返回它，溢出时则返回ULONG_MAX，字符串前面的空格、/、tab符被忽略

3.8.6 数学计算库函数

数学计算库函数的原型声明包含在头文件MATH.H中。表3.22列出了数学计算库函数的功能说明。

表3.22 数学计算库函数的功能说明

函数名及定义	功 能 说 明
int abs(int val); char cabs(char val); float fabs(float val); long labs(long val);	abs计算并返回val的绝对值，如果val为正，则不做改变就返回，如果为负，则返回相反数。 其余三个函数除了变量和返回值类型不同之外，其他功能完全相同
float exp(float x); float log(float x); float log10(float x);	exp计算并返回浮点数x的指数函数； log计算并返回浮点数x的自然对数（自然对数以e为底，e=2.718282）； log10计算并返回浮点数x以10为底x的对数
float sqrt(float x);	计算并返回x的正平方根
float cos(float x); float sin(float x); float tan(float x);	cos计算并返回x的余弦值； sin计算并返回x的正弦值； tan计算并返回x的正切值，所有函数的变量范围都是- $\pi/2 \sim +\pi/2$ ，变量的值必须在±65535之间，否则产生一个NaN错误

(续表)

函数名及定义	功 能 说 明
float acos(float x); float asin(float x); float atan(float x); float atan2(float y, float x);	acos计算并返回x的反余弦值; asin计算并返回x的反正弦值; atan计算并返回x的反正切值, 它们的值域为 $-\pi/2 \sim +\pi/2$; atan2计算并返回y/x的反正切值, 其值域为 $-\pi \sim +\pi$
float cosh(float x); float sinh(float x); float tanh(float x);	cosh计算并返回x的双曲余弦值; sinh计算并返回x的双曲正弦值; tanh计算并返回x的双曲正切值
float ceil(float x);	计算并返回一个不小于x的最小整数 (作为浮点数)
float floor(float x);	计算并返回一个不大于x的最大整数 (作为浮点数)
float modf(float x, float *ip);	将浮点数x分成整数和小数两部分, 两者都含有与x相同的符号, 整数部分放入*ip, 小数部分作为返回值
float pow(float x, float y);	计算并返回 x^y 的值, 如果x不等于0而y=0,则返回1。当x=0且y<=0或当x<0且y不是整数时, 则返回NaN

单片机片内资源应用

4.1 采用Keil C51编写应用程序的基本原则

C语言程序不要求具有固定格式，但我们在实际编写程序时还是应该遵守一定的规则。首先要采用清晰的书写风格。采用Keil C51编写单片机应用程序时，对于while、for、do-while、if-else、switch-case等语句，或这些语句的嵌套组合，应采用“缩进”的书写形式。对于复合语句或函数，通常需要使用花括号“{}”。当语句嵌套较多时，容易产生花括号不匹配的情况。μVision4的Edit下拉菜单中提供了一个“Goto Matching Brace”选项，将光标放在某个括号处，单击该选项，与之相匹配括号中的内容将反白显示，特别适用于检查各种括号的匹配情况。

对于一个表达式中各种运算执行的优先顺序不太明确或容易混淆的地方，应当采用圆括号“()”明确指定它们的优先顺序。对于程序中的函数，在使用之前应对函数的类型进行说明。对函数类型的说明必须保持与原来定义的函数类型相一致，不一致时将导致编译出错。对于具有返回值的函数，使用return语句时，最好使用括号“()”将被返回的内容括起来，这样可使程序执行过程更清晰，便于理解和维护。

在一般情况下，对于普通变量名或函数名采用小写字母表示，对于一些特殊变量名或由预处理命令#define所定义的常数，则采用大写字母表示。为了帮助理解和记忆，变量或函数名中可带有下画线，如ext_int0、data_max等，但是以下画线“_”开头的变量或函数名通常保留为C51编译系统所使用。为了避免混淆，不要将下画线用做变量或函数名的第一个字符。给变量或函数取名时，应按照“见名知义”的原则，如“ext_int0”表示外部中断函数、“data_max”表示最大数据值等。

数组与指针语句具有十分密切的关系。一个字符数组char * name= "hello"；可以采用数组形式name[0]或指针形式* name来表示字符串的第一个字母h，两者在意义上是完全相同的。在实际程序设计中，使用数组还是使用指针应视具体情况而定，一般来说，指针比较灵活简洁，而数组则比较直观，容易理解。

C语言是一种高级程序设计语言，与汇编语言不同。C语言提供了十分完备的规范化流程控制结构。因此在编写C51应用程序时，首先要注意尽可能采用结构化的程序设计方法，这样可使整个应用系统程序结构清晰，便于调试和维护。对于一个较大的应用程序，为了能

够集中精力考虑各种具体问题，通常将整个程序按功能分成若干个模块，不同模块完成不同的功能。各个模块程序可以分别编写，甚至可以由多个人分头编写。由于单个模块程序所完成的功能较为简单，程序的设计和调试也相应要容易一些。对于一些常用的功能模块，可以作为一个应用程序库，以便于以后直接调用。

用C语言进行模块化程序设计是比较容易实现的。一个C语言函数就可以认为是一个模块。所谓程序的模块化，不仅仅是要将整个程序划分成若干个功能模块，更重要的是还应当注意保持各个模块之间变量的相对独立性，即保持模块的独立性。在C语言的模块化编程过程中，如果过多地采用外部变量会减弱各个模块的独立性，因此为了保持整个程序具有较好的模块化结构，应尽量避免使用外部全局变量来传递数据信息，而应通过指定的参数来完成数据信息传递。对于不同的功能模块，可以分别指定相应的入口参数和出口参数，这样不会引起整个程序中变量管理的混乱。在 μ Vision4中很容易实现模块化编程，只要将分别编写的各个程序模块文件分别添加到项目中就可以了。

在程序设计过程中，对于经常使用的一些常数，如果将它们直接写到程序中去，一旦常数的数值发生变化，就必须逐个找出程序中所有对应的常数，逐一进行修改，这样必然会降低程序的可维护性和可移植性。因此为了便于对整个程序进行修改维护，或为了帮助记忆，应当采用预处理命令的方式来定义常数。对于一些常用的常数，如 π 、 e 、EOF、TRUE、FALSE及不同型号8051单片机中各种特殊功能寄存器和位地址等，应当集中起来放在一个头文件中进行定义，需要时再采用预处理命令#include将其包含到程序中去。这样做不仅可以提高编程效率，而且还可以避免输入错误。

一般来说，程序的执行效率主要取决于所采用算法的优劣和繁简。但对C语言而言，程序的执行效率在一定程度上还与程序的结构和设计方法有关。C语言具有十分丰富的运算符，合理地运用这些运算符可以设计出高效率的程序。例如，当条件表达式是由多个“&&”或“||”运算符连接在一起时，对于条件的判定总是从左至右逐个进行的，一旦条件满足时，就不再对后面其他条件进行判断。因此对于条件表达式的安排，应尽可能地将满足条件可能性较高的表达式放在整个条件式的前面。合理使用中间变量往往也可以提高程序的执行效率。

4.2 并行I/O端口

8051单片机具有4个并行I/O端口P0~P3，它们默认作为输入、输出端口使用，输入时数据可以缓冲，输出时数据可以锁存。P0~P3口都是准双向I/O端口，进行输入操作之前，应先向端口锁存器写入“1”，再进行读引脚操作。输入信号要求为TTL电平（低电平为0~0.8V，高电平为1.4~5V），如果实际输入信号达不到要求，就需要进行调理电路设计。各个I/O端口驱动能力有所不同，P0口输出时可以驱动8个TTL负载，P1~P3口则只能驱动4个TTL负载。另外，P0口用作输入、输出时，每个引脚都必须外加上拉电阻。

4.2.1 典型单片机输入、输出电路

对于较弱的输入开关信号，可通过如图4.1所示电路进行放大。图中，输入信号的频率

为10Hz、幅值为0~1V的开关量，放大后频率仍为10Hz，幅值达到0~5V。

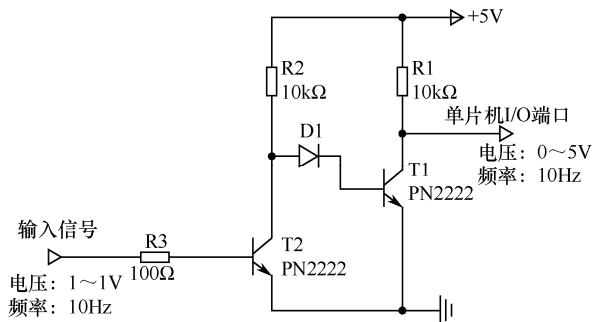


图4.1 弱信号输入电路

对于较强或不同电压等级的输入信号，可以采用光电隔离实现信号输入。图4.2为典型的强信号输入光电隔离电路。当强输入为低电平时，U1导通，连接到单片机I/O端口的光隔输出也为低电平；当强输入为高电平时，U1截止，连接到单片机I/O端口的光隔输出也为高电平，从而使强输入开关信号能够正确加载到单片机I/O端口。

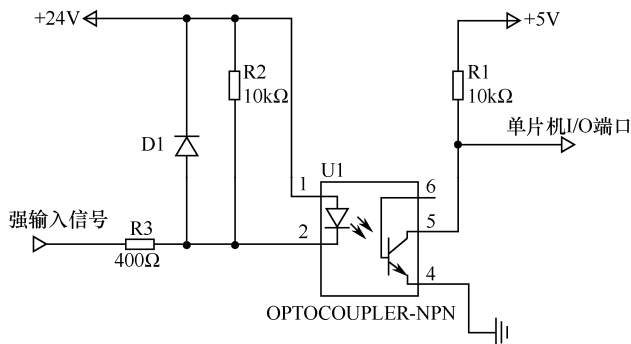


图4.2 强信号输入光电隔离电路

单片机自身驱动力能有限，一般只能驱动发光二极管、数码管等，对于电磁铁、继电器等功率器件则需要进行驱动电路设计。图4.3为单片机I/O端口经三极管耦合输出电路。T1、T2耦合后驱动T3。T1导通时，在R3、R4串联电路中有电流流过，从而使T2导通，T2提供了功率三极管T3的基极电流，T3的发射极用来驱动负载。R5一方面作为T2集电极的一个负载，另一方面T2截止时，T3基极所存储的电荷可以通过电阻R5迅速释放，加快T3的截止速度，有利于减小损耗。

图4.4为单片机I/O端口直接输出电路。集电极开路器件通过集电极负载电阻R1接到+15V电源，提升了驱动电压。但这种电路的开关速度较慢，若用来直接驱动功率管，则当后续电路具有电感性负载时，会导致功率管动态损耗加大。

为了改善I/O端口的开关速度，可采用如图4.5或图4.6所示的电路。图4.5具有快速开通功能，当I/O端口输出高电平时，输出点通过T1获得电压和电流，充电能力提高，从而加快开通速度；图4.6为推挽输出电路，不仅可提高开通速度，还可以提高带负载能力。

图4.7为单片机I/O端口驱动继电器的控制电路。其中，图4.7（a）具有电源变换作用，图4.7（b）具有光电隔离作用。

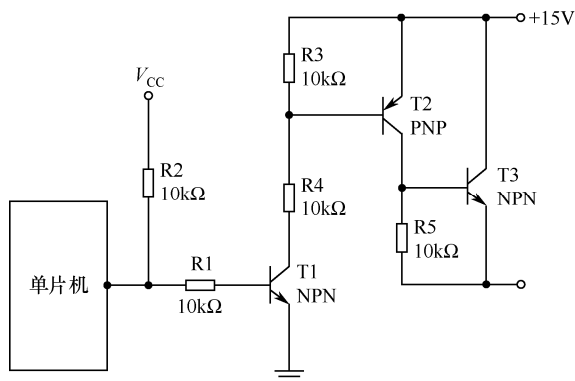


图4.3 单片机I/O端口经三极管耦合输出电路

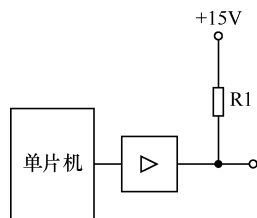


图4.4 单片机I/O端口直接输出电路

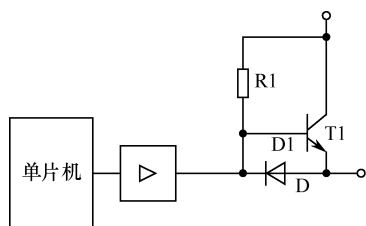


图4.5 单片机I/O端口快速开通输出电路

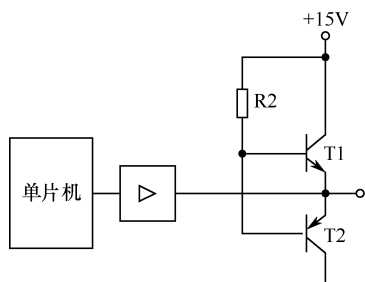
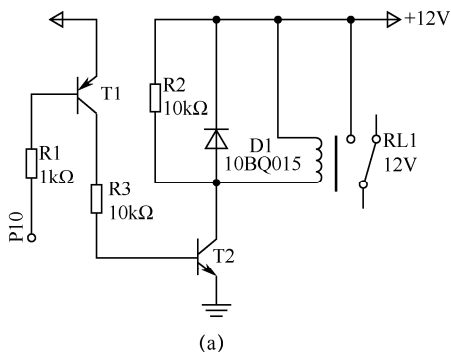
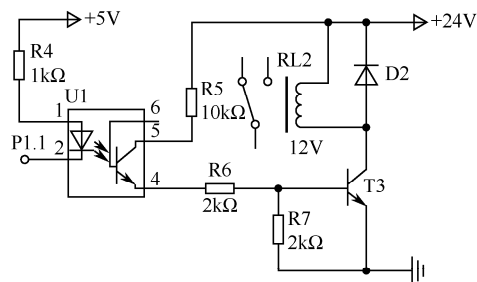


图4.6 单片机I/O端口推挽输出电路



(a)



(b)

图4.7 单片机I/O端口驱动继电器的控制电路

在采用继电器驱动的电路上，继电器的线圈电压宜高不宜低，如果继电器线圈电压过低，则由于三极管本身具有一定的压降，当三极管导通时，加在线圈两端的电压要减去三极管的压降，这样就难以使线圈导通。如线圈电压为5V，则三极管饱和导通时压降为0.7V，当三极管导通时，实际加在线圈两端的压降为4.3V。

4.2.2 单片机I/O端口应用编程

本节给出几个典型的单片机I/O端口应用编程的例子。

例4-1 通过P1口直接驱动发光二极管实现的流水灯。Proteus仿真电路如图4.8所示从

P0.0~P1.7分别输出低电平即可驱动发光二极管D1~D8反复轮流点亮。

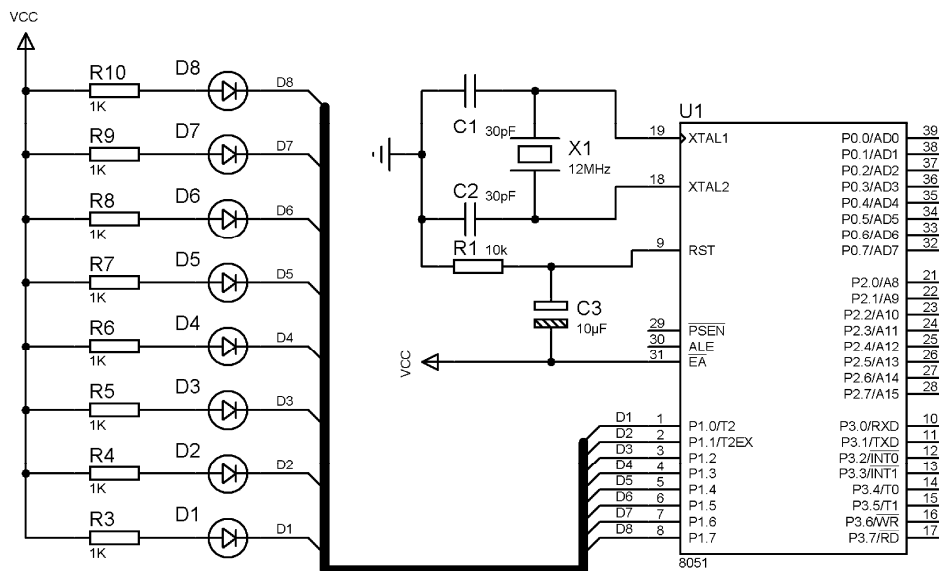


图4.8 P1口直接驱动发光二极管实现流水灯电路的Proteus仿真电路

C51源程序清单如下。

```
#include <reg52.h>
/***** 主函数 *****/
main() {
    unsigned char i,j;
    unsigned char LED;
    LED = 0x01;                //最低位LED点亮
    P1 = ~LED;                  //灌电流驱动
    while(1) {
        for(i=0;i<250;i++) {   //软件延时
            for(j=0;j<250;j++);
        }
        if(LED == 0x80) {      //如果流水到头，则折返到最低位点亮
            LED = 0x01;
        }
        else {
            LED = LED << 1;    //移位，行成流水灯
        }
        P1 = ~LED;
    }
}
```

例4-2 P2口通过三极管驱动发光二极管闪烁。Proteus仿真电路如图4.9所示。注意三极管T1、T2分别是PNP型三极管和NPN型三极管。它们的基极驱动电平不相同，因此P2.5输出低电平时将点亮发光二极管D1，P2.7输出高电平时将点亮发光二极管D2。

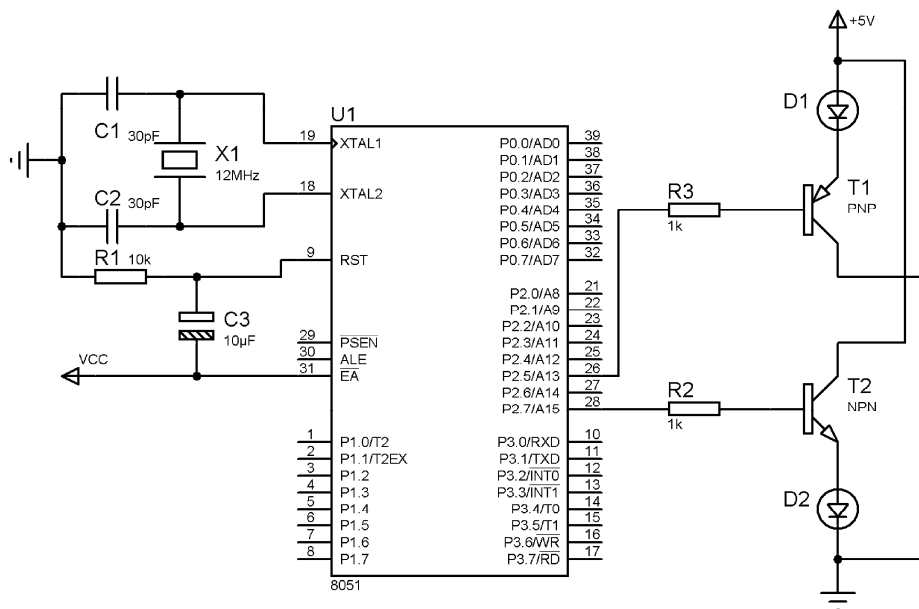


图4.9 P2口通过三极管驱动发光二极管闪烁的Proteus仿真电路

C51源程序清单如下。

```
#include <reg52.h>
sbit LED1 = P2 ^ 5;           //发光二极管
sbit LED2 = P2 ^ 7;

/***** 延时函数 *****/
void DelayMS(unsigned int ms){
    unsigned char i;
    while(ms--){
        for(i=0;i<120;i++);
    }
}

/***** 主函数 *****/
main(){
    while(1){
        LED1 = 1;              //D1熄灭
        LED2 = 0;              //D2熄灭
        DelayMS(100);
        LED1 = 0;              //D1点亮
        LED2 = 1;              //D2点亮
        DelayMS(100);
    }
}
```

例4-3 通过检测按键状态控制继电器驱动220V照明灯泡。Proteus仿真电路如图4.10所示。当K1按键被按下时，从P1.0输出高电平使三极管T1导通，继电器吸合驱动照明灯泡点

亮；当K2按键被按下时，从P1.0输出低电平使三极管T1截止，继电器断开，照明灯泡熄灭。

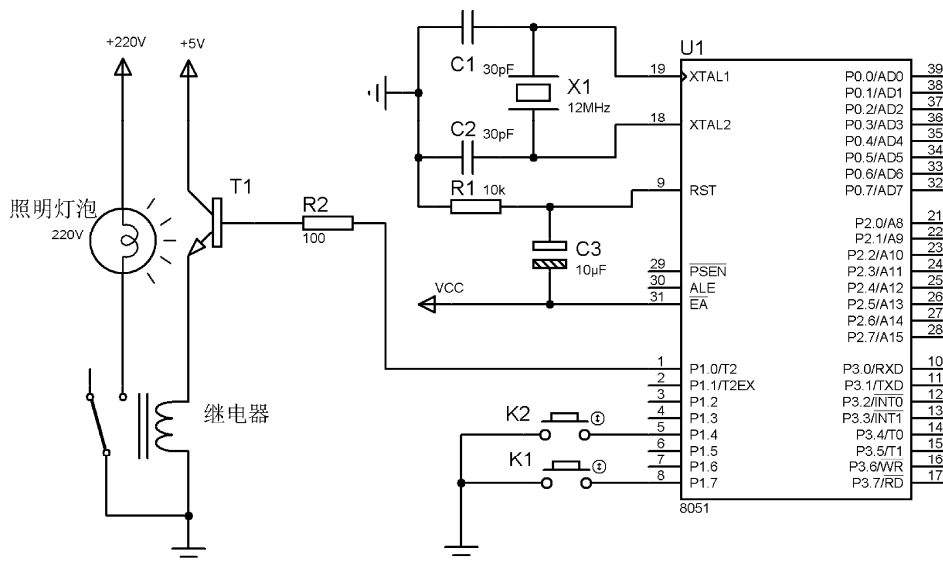


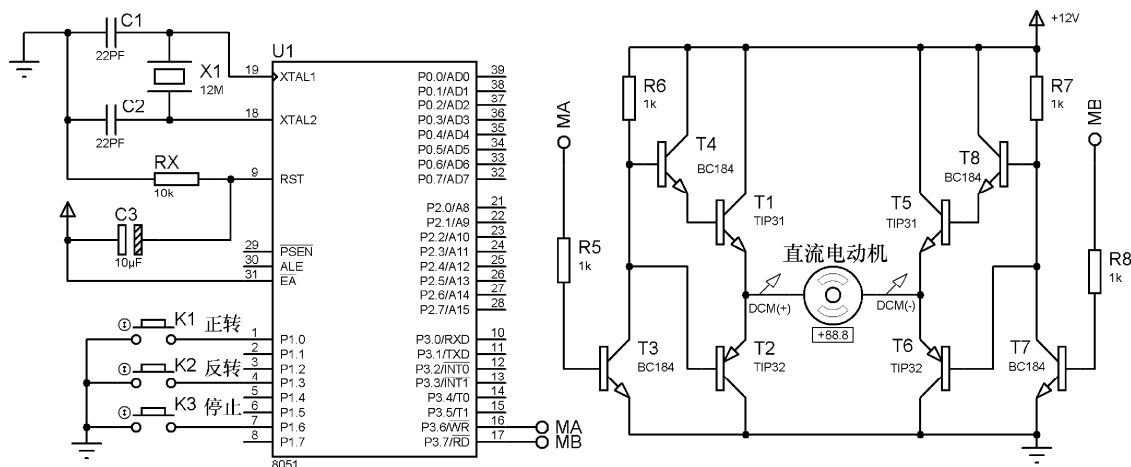
图4.10 通过检测按键状态控制继电器驱动220V照明灯泡的Proteus仿真电路

C51源程序清单如下。

```
#include <reg52.h>
sbit K1 = P1 ^ 7;
sbit K2 = P1 ^ 4;
sbit Relay = P1 ^ 0;
bit flg = 0;

/***** 主函数 *****/
main(void) {
    Relay = 0;          //继电器断开
    while(1) {
        if(K1 == 0) {   //检测按键K1是否按下
            while(K1==0); //等待按键释放
            Relay = 1;    //继电器吸合
        }
        if(K2 == 0) {   //检测按键K2是否按下
            while(K1==0); //等待按键释放
            Relay = 0;    //继电器断开
        }
    }
}
```

例4-4 通过检测按键状态控制直流电动机的正、反转。Proteus仿真电路如图4.11所示。由三极管T1~T8组成直流电动机的对称控制电路。单片机系统复位后，I/O端口默认输出高电平，三极管T1~T8均处于导通状态，直流电动机两端电压相同，电动机停止旋转。当K1按键被按下时，P3.6输出低电平，三极管T2、T3截止，直流电动机DCM(+)端电压高于直流电动



C51源程序清单如下。

```
#include <reg52.h>

sbit K1    = P1^0;
sbit K2    = P1^3;
sbit K3    = P1^6;
sbit MA    = P3^6;
sbit MB    = P3^7;

/***** 主函数 *****/
void main(void){
    while(1){
        if(K1 == 0){
            while(K1 == 0);
            MA    = 0;
            MB    = 1;
        }
        if(K2 == 0){
            while(K1 == 0);
            MA    = 1;
            MB    = 0;
        }
        if(K3 == 0){
            while(K1 == 0);
            MA    = 1;
            MB    = 1;
        }
    }
}
```



```

    }
}

```

例4-5 通过检测按键状态控制步进电动机的正、反转。Proteus仿真电路如图4.12所示。通过单片机四根I/O口线P1.0~P1.3的输出完成环形脉冲分配，实现对四相步进电动机每组线圈中电流的顺序切换，使电动机做步进式旋转，采用单、双八拍控制方式，同时采用三个按键K1、K2、K3来控制电动机的正转和反转。由于单片机I/O端口的驱动能力不够，使用达林顿驱动器ULN2003来增加P1口的驱动电流，ULN2003可以在5V工作电压下直接驱动TTL、CMOS或继电器等负载，适用于各种大功率驱动系统。

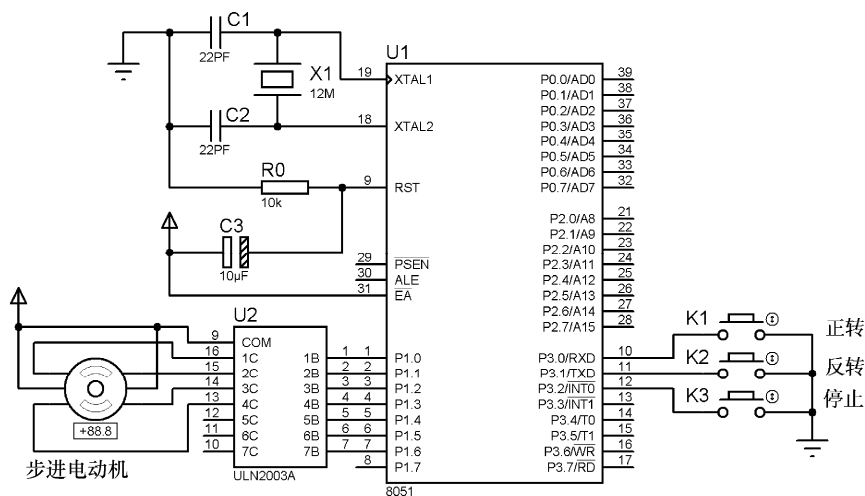


图4.12 通过检测按键状态控制步进电动机正、反转的Proteus仿真电路

C51源程序清单如下。

```

#include <reg52.h>
#define uint unsigned int
#define uchar unsigned char
sbit K1 = P3^0;
sbit K2 = P3^1;
sbit K3 = P3^2;
uchar code FFW[]={0x01,0x03,0x02,0x06,0x04,0x0c,0x08,0x09};
uchar code REV[]={0x09,0x08,0x0c,0x04,0x06,0x02,0x03,0x01};

/***** 延时函数 *****/
void DelayMS(uint ms){
    uchar i;
    while(ms--){
        for(i=0;i<120;i++);
    }
}

/***** 正转函数 *****/
void SETP_MOTOR_FFW(void){

```

```

    uchar j;
    while(1){
        for(j=0;j<8;j++){
            P1 = FFW[j];
            DelayMS(20);
        }
        if(K3 == 0) break;
    }
}

/***** 反转函数 *****/
void SETP_MOTOR_REV(void){
    uchar j;
    while(1){
        for(j=0;j<8;j++){
            P1 = REV[j];
            DelayMS(20);
        }
        if(K3 == 0) break;
    }
}

/***** 主函数 *****/
void main(){
    while(1){
        if(K1 == 0){
            SETP_MOTOR_FFW();
        }
        if(K2 == 0){
            SETP_MOTOR_REV();
        }
    }
}

```

4.3 中断系统

单片机与外部设备之间的数据交换可以采用两种方式，即查询方式和中断方式。查询方式传送数据也称为条件传送，通过查询确信外设已处于“准备好”状态，单片机才发出访问外设的指令。查询方式的优点是通用性好，缺点是需要有一个等待查询过程，CPU在等待查询期间不能进行其他操作，从而导致单片机的工作效率降低。

中断方式传送数据具有可以有效提高单片机工作效率，适合于实时控制系统等优点，因而更为常用。当CPU正在处理某件事情的时候，外部发生的某一事件（如电平的改变、脉冲边沿跳变、定时器/计数器溢出等）请求CPU迅速去处理，于是CPU暂时中断当前的工作，转去处理所发生的事件。处理完该事件以后，再回到原来被中断的地方继续原来的工作。这样的过

程被称为中断。中断流程如图4.13所示。

与查询方式不同，中断方式是外设主动提出数据传送的请求，CPU在收到这个请求以前，一直在执行主程序，只是在收到外设希望进行数据传送的请求之后，才中断原有主程序的执行，暂时去与外设交换数据，数据交换完毕立即返回主程序继续执行。中断方式完全消除了CPU在查询方式中的等待现象，大大提高了CPU的工作效率。

8051单片机可以接受的中断申请一般不止一个，对于这些不止一个的中断源进行管理，就是中断系统的任务。这些任务一般包括以下几方面。

1. 开中断或关中断

中断的开放或关闭可以通过指令对相关特殊功能寄存器的操作来实现，这是CPU能否接受中断申请的关键，只有在开中断的情况下，才有可能接受中断源的申请。

2. 中断排队

8051单片机是一个多中断源系统，在开中断的条件下，如果有若干个中断申请同时发生，就需要决定先对哪一个中断申请进行响应，这就是中断排队的问题，也就是要对各个中断源做一个优先级排序，单片机先响应优先级别高的中断申请。

3. 中断响应

单片机在响应了中断源的申请时，应使CPU从主程序转去执行中断服务子程序，同时要把断点地址送入堆栈进行保护，以便在执行完中断服务子程序后能返回到原来的断点继续执行主程序，断点地址入栈是由单片机内部硬件自动完成的，中断系统还要能确定各个被响应中断源的中断服务子程序的入口。

4. 中断撤除

在响应中断申请以后，返回主程序之前，中断申请应该撤除，否则就等于中断申请仍然存在，这将影响对其他中断申请的响应。8051单片机内部硬件只能对一部分中断申请在响应之后自动撤除，这一点在使用中一定要注意。

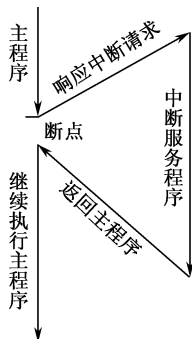


图 4.13 中断流程

4.3.1 中断系统结构与中断控制

8051单片机的中断系统结构如图4.14所示。

8051单片机是个多中断源系统，有5个中断源，即两个外部中断、两个定时器/计数器中断及1个串行口中断。

两个外部中断分别从 $\overline{\text{INT0}}$ (P3.2) 和 $\overline{\text{INT1}}$ (P3.3) 引脚输入。外部中断请求信号可以有两种方式，即电平触发方式和负边沿触发方式。若是电平触发方式，则要在 $\overline{\text{INT0}}$ 或 $\overline{\text{INT1}}$ 引脚上检测到低电平信号即为有效的中断申请。若是负边沿触发方式，则需在 $\overline{\text{INT0}}$ 或 $\overline{\text{INT1}}$ 引脚上检测到从1到0的负边沿跳变，才属于有效申请。

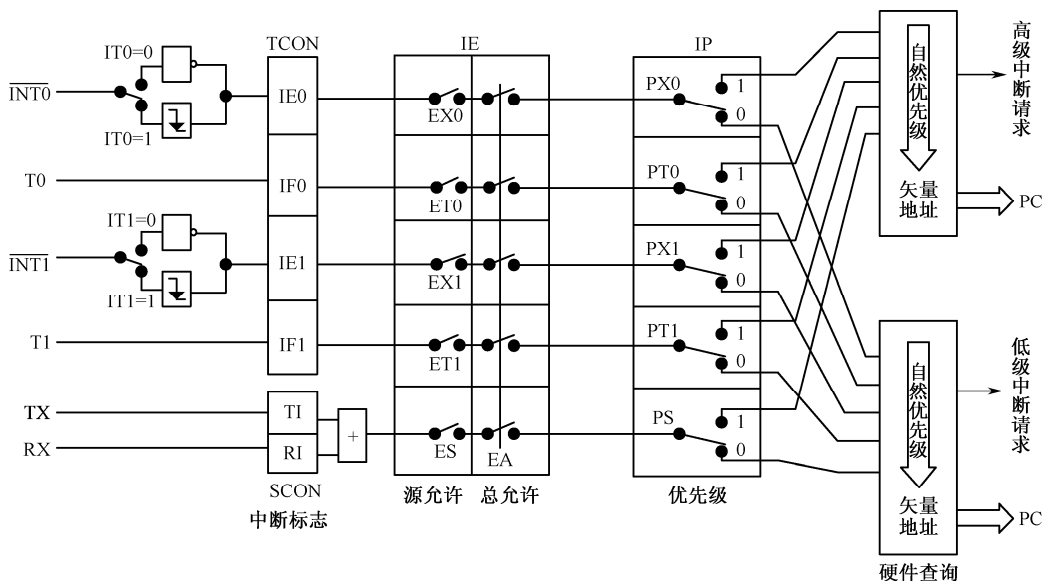


图4.14 8051单片机的中断系统结构

两个定时器/计数器中断是当T0或T1溢出（由全“1”进入全“0”）时发出的中断申请，属于内部中断。

串行口中断也属于内部中断，是在串行口接收或发送完一组串行数据后自动发出的中断申请。

从用户的角度来看，8051单片机的中断系统就是如下几个特殊功能寄存器：

- 定时器控制寄存器TCON；
- 中断允许寄存器IE；
- 中断优先级寄存器IP；
- 串行口控制寄存器SCON。

通过对以上各特殊功能寄存器中相应位的置1或清0，即可实现各种中断控制功能。

CPU在检测到有效的中断申请后，使某些相应的标志位置1，CPU在下一个机器周期检测这些标志以决定是否要响应中断。这些标志位分别对应于特殊功能寄存器TCON和SCON的相应位。

TCON寄存器的地址为88H，其中各位都可以位寻址，位地址为88H~8FH。TCON寄存器中与中断有关的各控制位分布为

D7	D6	D5	D4	D3	D2	D1	D0
TF1		TF0		IE1	IT1	IE0	IT0

各控制位的含义为：

- IT0：选择外中断 $\overline{\text{INT0}}$ 的中断触发方式。IT0=0为电平触发方式，低电平有效。IT0=1为负边沿触发方式， $\overline{\text{INT0}}$ 脚上的负跳变有效。IT0的状态可以用指令来置1或清0。
- IE0：外中断 $\overline{\text{INT0}}$ 的中断申请标志。当检测到 $\overline{\text{INT0}}$ 上存在有效中断申请时，由内部硬件使IE0置1。当CPU转向中断服务时，由内部硬件将IE0清0。
- IT1：选择外中断 $\overline{\text{INT1}}$ 的触发方式，功能与IT0类似。
- IE1：外部中断 $\overline{\text{INT1}}$ 的中断申请标志，功能与IE0相同。

- **TF0**: 定时器/计数器T0溢出中断申请标志。当T0溢出时, 由内部硬件将TF0置1, 当CPU转向中断服务时, 由内部硬件将TF0清0。
- **TF1**: 定时器1溢出中断申请标志, 功能与TF0相同。
- 外部中断和定时器/计数器溢出中断的申请标志, 在CPU响应中断之后能够自动撤除。

8051单片机串行口的中断申请标志位于特殊功能寄存器SCON中。SCON寄存器的地址为98H, 其中各位都可以位寻址, 位地址为98H~9FH。串行口的中断申请标志只占用SCON中的两位, 分布为

D7	D6	D5	D4	D3	D2	D1	D0
						TI	RI

各控制位的含义为:

- **RI**: 接收中断标志, 当接收完一帧串行数据后置1, 必须由软件清0。
- **TI**: 发送中断标志。当发送完一帧串行数据后置1, 必须由软件清0。

串行口的中断申请标志是由TI和RI相或以后产生的, 并且串行口中断申请在得到CPU响应之后不会自动撤除, 必须通过软件程序撤除。

8051单片机中断的开放和关闭是由特殊功能寄存器IE来实现两级控制的。所谓两级控制是指在寄存器IE中有一个总允许位EA, 当EA=0时, 就关闭了所有的中断申请, CPU不响应任何中断申请。而当EA=1时, 对各中断源的申请是否开放, 还要看各中断源的中断允许位的状态。

中断允许寄存器IE的地址为A8H, 其中各位都可以位寻址, 位地址为A8H~AFH。总允许位EA和各中断源允许位在IE寄存器中的分布为

D7	D6	D5	D4	D3	D2	D1	D0
EA			ES	ET1	EX1	ET0	EX0

各控制位的含义为:

- **EX0**: 外部中断0 ($\overline{\text{INT0}}$) 的中断允许位。EX0=1, 允许中断; EX0=0, 不允许中断。
- **ET0**: 定时器/计数器T0的溢出中断允许位。ET0=1, 允许中断; ET0=0, 不允许中断。
- **EX1**: 外部中断1 ($\overline{\text{INT1}}$) 的中断允许位。ET1=1, 允许外部中断1申请中断; EX1=0则不允许中断。
- **ET1**: 定时器/计数器T1的溢出中断允许位。ET1=1, 允许T1溢出中断; ET1=0, 则不允许T1溢出中断。
- **ES**: 串行口中断源允许位。ES=1, 串行口开中断; ES=0, 串行口关中断。
- **EA**: 中断总允许位。EA=0时, CPU关闭所有的中断申请, 只有EA=1时, 才能允许各个中断源的中断申请, 但还要取决于各中断源中断允许控制位的状态。

8051单片机在复位时, IE各位的状态都为0, 所以CPU是处于关中断的状态。对于串行口来说, 其中断请求在被响应之后, CPU不能自动清除其中断标志, 在这些情况下要注意用指令来实现中断的开放或关闭, 以便进行各种中断处理。

8051单片机的中断系统具有两个中断优先级, 对于每一个中断请求源可编程为高优先级或低优先级中断, 以实现两级中断嵌套。每个中断源的优先级分别由特殊功能寄存器IP来管理。

IP寄存器的地址为B8H, 其中各控制位是可以位寻址的, 位地址为B8H~BCH。IP寄存器中各控制位分布为

D7	D6	D5	D4	D3	D2	D1	D0
			PS	PT1	PX1	PT0	PX0

各位的含义为：

- PX0：外部中断 $\overline{\text{INT0}}$ 中断优先级控制位。
- PT0：定时器/计数器T0中断优先级控制位。
- PX1：外部中断 $\overline{\text{INT1}}$ 中断优先级控制位。
- PT1：定时器/计数器T1中断优先级控制位。
- PS：串行口中断优先级控制位。

IP寄存器中若某一个控制位置1，则相应的中断源就规定为高优先级中断；反之，若某一个控制位置0，则相应的中断源就规定为低优先级中断。一个正在执行的低优先级中断服务程序能被高优先级中断源的中断申请所中断，形成中断嵌套，如图4.15所示。相同级别的中断源不能相互中断其服务程序，也不能被另一个低优先级的中断源所中断。若CPU正在执行高优先级的中断服务子程序，则不能被任何中断源所中断。

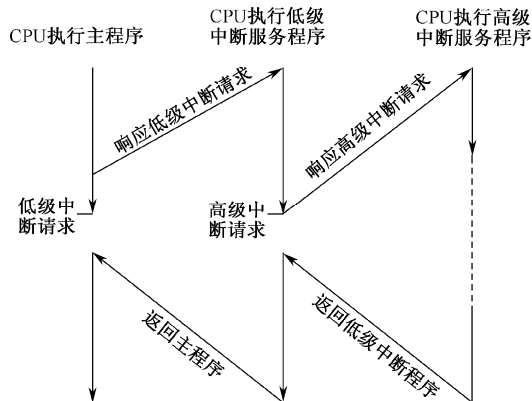


图4.15 中断嵌套

4.3.2 中断响应

当有某个中断源请求中断，同时特殊功能寄存器IE中相应控制位处于置1状态时，CPU就可以响应中断。8051单片机有5个中断源，但只有两个中断优先级，因此必然会有若干个中断源处于同样的中断优先级。当两个同样级别的中断申请同时到来时，CPU应该如何响应呢？在这种情况下，8051单片机内部有一个固定的查寻次序，当出现同级中断申请时，就按这个次序来处理中断响应。8051单片机的5个中断源及其同级内的优先级次序见表4.1。

表4.1 8051单片机的中断源

中 断 源	入 口 地 址	同级内的优先级顺序	说 明
外部中断0	0003H	最高	来自P3.2引脚（ $\overline{\text{INT0}}$ ）的外部中断请求
定时器/计数器T0	000BH		定时器/计数器T0溢出中断请求
外部中断1	0013H		来自P3.3引脚（ $\overline{\text{INT1}}$ ）的外部中断请求
定时器/计数器T1	001BH		定时器/计数器T1溢出中断请求
串行口	0023H	最低	串行口完成一帧数据的发送或接收中断

表4.1中列出的只是8051单片机的5个最基本中断源。不同型号单片机除了这5个基本的中断源之外，还有它们各自专有的中断源，如8052就还有一个定时器/计数器T2溢出中断，T2的中断入口地址为002BH。

8051单片机在接收到中断申请以后，先把这些申请锁定在各自的中断标志位中，然后在下一个机器周期按表4.1规定的内部优先顺序和中断优先级分别来查询这些标志，并在一个机器周期之内完成检测和优先排队。响应中断的条件有3个：

① 必须没有同级或更高级别的中断正在得到响应，如果有，则必须等CPU为它们服务完毕，返回主程序并执行一条指令之后才能响应新的中断申请；

② 必须要等当前正在执行的指令执行完毕以后，CPU才能响应新的中断申请；

③ 若正在执行的指令是RETI（中断返回）或是任何访问IE寄存器或IP寄存器的指令，则必须要在执行完该指令及紧随其后的另外一条指令之后，才可以响应新的中断申请。在这种情况下，响应中断所需的时间就会加长。这个响应条件是8051单片机所特有的。

若上述条件满足，CPU就在下一个机器周期响应中断，完成两项工作：一项是把中断点的地址，即当前程序计数器PC的内容送入堆栈保护；另一项是根据中断的不同来源把程序的执行转移到相应的中断服务子程序的入口。在8051单片机中，这种转移关系是固定的，对于每一种中断源，都有一个固定的中断服务子程序入口地址，见表4.1。

CPU响应中断的时候，中断请求被锁存在TCON和SCON的标志位。当某个中断请求得到响应之后，相应的中断标志位应该予以清除（即清0），否则CPU又会继续查询这些标志位而认为又有新的中断申请来到，实际上这种中断申请并不存在。因此就存在一个中断请求的撤除问题。8051单片机有5个中断源，对于其中的两种，在响应之后，系统能通过硬件自动使标志位清0（即撤除），它们是：

- 定时器0或1的中断请求标志TF0或TF1；
- 外部中断0或1的中断请求标志IE0或IE1。

在这里需要注意的是外部中断。由于外部中断有两种触发方式，即低电平方式和负边沿方式，因此对于边沿触发方式比较简单，因为在清除IE0或IE1以后，必须再来一个负边沿信号，才可能使标志位重新置1。对于低电平触发方式则不同，若仅是由硬件清除IE0或IE1标志，而加在 $\overline{\text{INT0}}$ 或 $\overline{\text{INT1}}$ 引脚上的低电平不撤销，则在下一个机器周期CPU检测外中断申请时会发现又有低电平信号加在外中断输入上，又会使IE0或IE1置1，从而产生错误的结果。8051单片机的中断系统没有对外的联络信号，即中断响应之后没有输出信号去通知外设结束中断申请，因此必须由用户自己来关心和处理这个问题。

对于串行口的中断请求标志TI和RI，中断系统不予以自动撤除。在响应串行口中断之后要先测试这两个标志位，以决定是接收还是发送，故不能立即撤销。但在使用完毕之后应使之清0，以结束这次中断申请。TI和RI的清0操作可在中断服务子程序中用指令来实现。

8051单片机在响应中断之前，必须对中断系统进行初始化，也就是对组成中断系统的若干个特殊功能寄存器中的各控制位赋值。中断系统的初始化一般需要完成以下操作：

- 开中断；
- 确定各中断源的优先级；
- 若是外部中断，则应规定是低电平触发还是负边沿触发。

CPU响应中断后将转到中断源的入口地址开始执行中断服务程序。8051单片机的每个中断源都有其固定的入口地址，它们的处理过程也有所区别。在一般情况下，中断处理包括两

个部分：一是保护现场；二是为中断服务。

所谓保护现场，就是将需要在中断服务程序中使用而又不希望破坏其中原来内容的工作寄存器压入堆栈中保护起来，等中断服务完成后，再从堆栈中弹出以恢复原来的内容。通常需要保护的寄存器有PSW、A及其他工作寄存器。

处理中断时要注意以下几点。

① 8051各中断源的入口地址之间仅相隔8个单元，如果中断服务程序的长度超过8个地址单元时，则应在中断入口地址处安排一条转移指令，转到其他有足够空余存储器单元的地址空间。

② 若在执行当前中断服务程序时需要禁止更高级中断源，则要用软件指令关闭中断，在中断返回之前再开放中断。

③ 在保护和恢复现场时，为了不使现场信息受到破坏或造成混乱，保护现场之前应先关中断，若需要允许高级中断，则应在保护现场之后再开中断。同样，在恢复现场之前，也应先关中断，恢复现场之后再开中断。

④ 及时清除那些不能被硬件自动清0的中断请求标志，以免产生错误的中断。

⑤ 编写中断服务函数。Keil C51编译器中规定中断服务函数的格式为

```
void 函数名(void) [interrupt n] [using m]
```

关键字interrupt后面的 n 是中断号，对于8051单片机， n 的取值范围为0~4，编译器根据中断号自动计算出对应中断源的入口地址。

关键字using后面的 m 是该中断函数所使用的工作寄存器区，默认为当前工作寄存器区。

特别需要注意的是，中断服务函数既没有返回值，也没有调用参数，因此任何时候中断服务函数都不能被其他函数调用。

最后说明一下中断的响应时间问题，CPU并不是在任何情况下都对中断请求立即响应，在不同情况下，中断响应的的时间有所不同，下面以外部中断为例来进行说明。

外部中断请求在每个机器周期的S5P2期间，经过反向后锁存到IE0或IE1标志中，CPU在下一个机器周期才会查询这些标志，这时如果满足响应中断的条件，则CPU响应中断时，需要执行一条两个机器周期的调用指令，以转到相应的中断服务程序入口。这样从外部中断请求有效到开始执行中断服务程序的第一条指令，至少需要3个机器周期。

如果在申请中断时，CPU正在执行最长的指令（如乘、除指令），则额外等待时间增加3个机器周期；若正在执行中断返回（RETI）或访问IE、IP寄存器的指令，则额外等待时间又要增加两个机器周期。综合估算，若系统中只有一个中断源，则中断响应时间为3~8个机器周期。

4.3.3 中断系统应用编程

8051单片机只有两个外部中断源 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ ，当实际应用中需要多个外部中断源时，可采用硬件请求和软件查询相结合的办法进行扩展，把多个中断源通过“或非”门接到外部中断输入端，同时又连到某个I/O端口，这样每个中断源都能引起中断，然后在中断服务程序中通过查询I/O端口的状态来区分是由哪个中断源引起的中断。若有多个中断源同时发出中断请求，则查询的次序就决定了同一优先级中断中的优先级。

例4-6 利用中断查询扩展中断源。

Proteus仿真电路如图4.16所示。3个转换开关SW1~SW3通过一个或非门连到8051的外中断输入引脚 $\overline{\text{INT0}}$ ，按键B1连到8051的外中断输入引脚 $\overline{\text{INT1}}$ 。SW1~SW3的初始位置接地，当SW1~SW3中无论哪个转换到高电平时都会使 $\overline{\text{INT0}}$ 引脚电平变低，向CPU提出中断申请，究竟是哪个转换开关提出中断申请，可以在 $\overline{\text{INT0}}$ 中断服务程序中通过查询P1.0、P1.2、P1.4的逻辑电平获知，同时单片机通过P1.1、P1.3、P1.5输出高电平点亮相应的LED指示灯。当按键B1被按下（接地）时，将触发外部中断 $\overline{\text{INT1}}$ ，在 $\overline{\text{INT1}}$ 中断服务程序中向P1口输出低电平，熄灭所有LED指示灯。

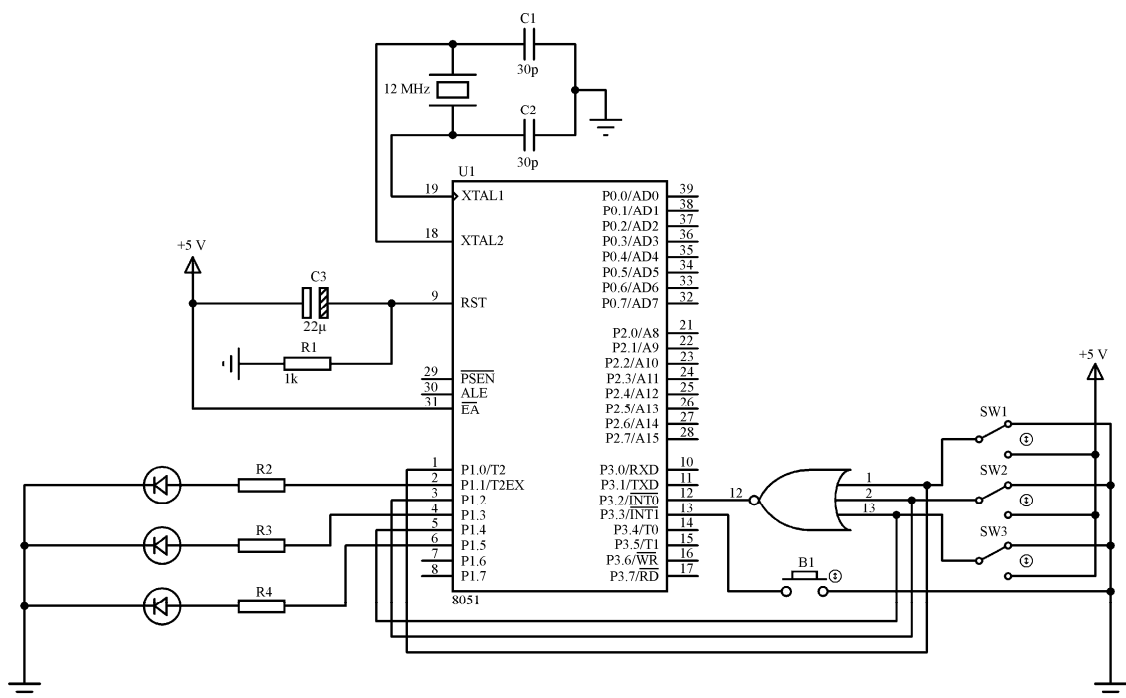


图4.16 利用中断查询扩展中断源的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int

sbit K1=P1^0;
sbit K2=P1^2;
sbit K3=P1^4;
sbit L1=P1^1;
sbit L2=P1^3;
sbit L3=P1^5;

/***** INT0中断服务函数 *****/
void int0() interrupt 0 {
    if(K1==1) L1=1;
```

```

        if(K2==1) L2=1;
        if(K3==1) L3=1;
    }

    /***** INT1中断服务函数 *****/
    void int1() interrupt 2 {
        P1&=0x55;
    }

    /***** 主函数 *****/
    void main(){
        P1&=0x55;
        IE=0x85; TCON=0x05;
        while(1);
    }

```

上面这个例子比较简单，不需要保护现场，在实际应用时，如果中断服务程序较复杂，需要采用多个工作寄存器时，一定要注意现场的保护和恢复。

8051单片机的中断系统具有两个优先级，每个中断源都可以设置为高、低优先级，多个中断同时发生时，CPU根据优先级的高、低分先后进行响应，并执行相应的中断服务程序。一个正在执行的低优先级中断服务程序能被高优先级中断源的中断申请所中断，形成中断嵌套。相同级别的中断源不能相互中断，也不能被另一个低优先级的中断源所中断。若CPU正在执行高优先级的中断服务子程序，则不能被任何中断源所中断。

例4-7 高、低优先级中断嵌套。

Proteus仿真电路如图4.17所示。在8051单片机外部中断INT0、INT1端分别通过两个按键接地，单片机的P0、P1、P2口分别接3个共阳极LED数码管。将INT1设置为高优先级，INT0设置为低优先级，负边沿触发。

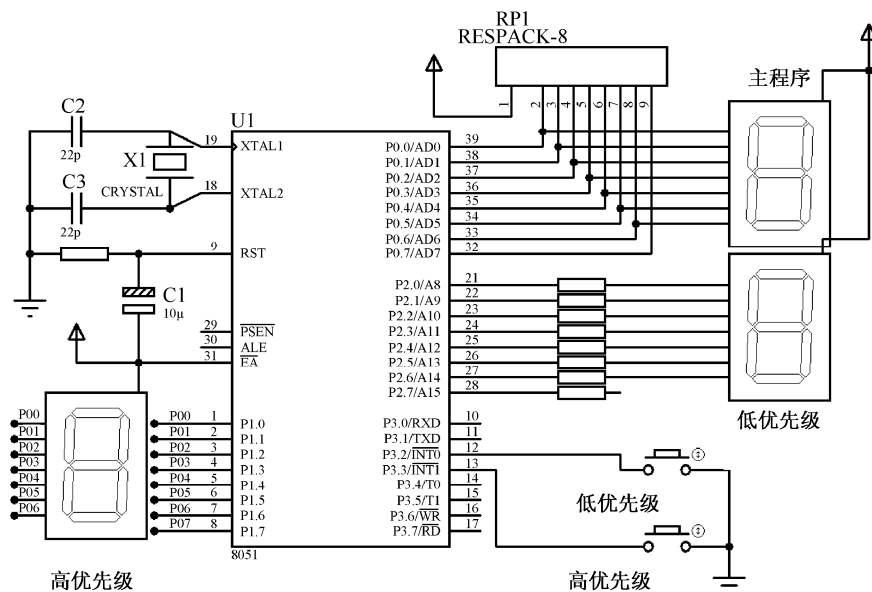


图4.17 高、低优先级中断服务程序嵌套的Proteus仿真电路

主程序在开中断后进入循环状态,通过P0口循环显示“1”~“8”字符,此时无论按下“低优先级”或“高优先级”按键,主程序都会被中断,进入中断服务程序,通过P2或P1口显示“1”~“8”字符。

如果先按下“低优先级”按键,则P0口的显示将停在某一数字,进入低优先级中断服务程序,通过P2口显示“1”~“8”字符;在P2口显示结束之前按下“高优先级”按键,则P2口的显示将停在某一数字,进入高优先级中断服务程序,通过P1口显示“1”~“8”字符,高优先级中断服务程序结束后,先返回到低优先级中断服务程序继续执行,即P2口从刚才暂停的数字继续显示,P2口显示结束后返回到主程序执行,即P0口从刚才暂停的数字继续循环显示。

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int

uchar seg[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80}; //LED段码表

sbit K1=P3^2; //定义按键
sbit K2=P3^3;

/***** 延时函数 *****/
void delay(){
    uint j;
    for(j=0;j<31000;j++);
}

/***** INT0中断服务函数 *****/
void int0() interrupt 0 using 1{
    uchar i;
    for(i=1;i<9;i++){
        P2=seg[i];delay(); //循环显示1~8
    }
    P2=0xFF;
}

/***** INT1中断服务函数 *****/
void int1() interrupt 2 using 2{
    uchar i;
    for(i=1;i<9;i++){
        P1=seg[i];delay(); //循环显示1~8
    }
    P1=0xFF;
}

/***** 主函数 *****/
void main(){
    uchar i;
    IE=0x85; TCON=0x05; PX1=1; //开中断,设置INT1为高优先级
```

```

while(1){
    for(i=1;i<9;i++){
        P0=seg[i];delay(); //循环显示1~8
    }
}
}

```

4.4 定时器/计数器

8051单片机内部有两个16位可编程定时器/计数器，记为T0和T1。8052单片机内除了T0和T1之外，还有第三个16位的定时器/计数器，记为T2。它们的工作方式可以通过指令对相应的特殊功能寄存器编程来设定，或用于定时器，或用于外部事件计数器。

定时器/计数器在硬件上由双字节加法计数器TH和TL组成。用于定时器时，计数脉冲由单片机内部振荡器提供，计数频率为 $f_{osc}/12$ ，每个机器周期加1。用于计数器时，计数脉冲由P3口的P3.4（或P3.5），即T0（或T1）引脚输入，外部脉冲的下降沿触发计数，计数器在每个机器周期的S5P2期间采样外部脉冲，若一个周期的采样值为1，下一个周期的采样值为0，则计数器加1，故识别一个从0到1的跳变需要两个机器周期，所以对外部计数脉冲的最高计数频率为 $f_{osc}/24$ ，同时还要求外部脉冲的高、低电平保持时间均要大于一个机器周期。

4.4.1 定时器/计数器的工作方式与控制

8051单片机定时器/计数器的工作方式由特殊功能寄存器TMOD编程决定，定时器/计数器的启动运行由特殊功能寄存器TCON编程控制。无论用于定时器还是用于计数器，每当产生溢出时，都会向CPU发出中断申请。

方式控制寄存器TMOD的地址为89H，控制字格式为

D7	D6	D5	D4	D3	D2	D1	D0
GATE	C/\overline{T}	M1	M0	GATE	C/\overline{T}	M1	M0
←—— T1方式字段 ——→				←—— T0方式字段 ——→			

其中，低4位为T0的控制字，高4位为T1的控制字。各控制位的含义如下：

- GATE为门控位。它对定时器/计数器的启动起辅助控制作用。GATE=1时，定时器/计数器的计数受外部引脚P3.2（ $\overline{INT0}$ ）或P3.3（ $\overline{INT1}$ ）输入电平的控制，此时，只有当P3口的P3.2（或P3.3）引脚，即 $\overline{INT0}$ （或 $\overline{INT1}$ ）上的电平为1才能启动计数；GATE=0时，定时器/计数器的运行不受外部引脚输入电平的控制。
- C/\overline{T} 为方式选择位。 $C/\overline{T}=0$ 为定时器方式，采用单片机内部振荡脉冲的12分频信号作为计数脉冲，若采用12 MHz的晶振，则计数频率为1MHz，从计数值便可计算出定时时间。 $C/\overline{T}=1$ 为计数器方式，采用外部引脚（T0使用P3.4，T1使用P3.5）的输入脉冲作为计数脉冲，当T0（或T1）上的输入信号发生从高到低的负跳变时，计数器加1。最高计数频率为单片机晶振频率的1/24。
- M1、M0两位的状态确定定时器/计数器的工作方式，详见表4.2。

表4.2 定时器/计数器的方式选择

M1	M0	工作方式
0	0	方式0, 为13位定时器/计数器
0	1	方式1, 为16位定时器/计数器
1	0	方式2, 为自动重装常数的8位定时器/计数器
1	1	方式3, 仅适用于T0, 分成两个8位定时器/计数器

运行控制寄存器TCON的地址为88H, 格式为

D7	D6	D5	D4	D3	D2	D1	D0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

各控制位的含义如下:

- TF1为定时器/计数器T1的溢出标志位。当T1被允许计数以后, T1从初值开始加1计数, 计数器的最高位产生溢出时置“1”TF1, 并向CPU申请中断, 当CPU响应中断时, 由硬件清“0”TF1。
- TR1为定时器/计数器的运行控制位, 由软件置位和复位。当方式控制寄存器TMOD中的GATE位为0, 且TR1为1时允许T1计数, TR1为0时禁止T1计数。当GATE为1时, 仅当TR1为1且 $\overline{\text{INT1}}$ (P3.2)输入为高电平时才允许T1计数, 当TR1为0或 $\overline{\text{INT1}}$ 输入为低电平时都禁止T1计数。
- TR0为定时器T0的运行控制位, 其功能与TR1类似。
- TF0为定时器T0的溢出标志位, 其功能与TF1类似。

运行控制寄存器TCON的低4位与外部中断有关, 已在4.3节中介绍, 这里不再赘述。

下面以定时器/计数器T1为例介绍其工作方式。

1. 方式0和方式1

方式0为13位定时器/计数器, 由TL1的低5位和TH1的8位构成。方式1为16位定时器/计数器, TL1和TH1均为8位。图4.18为方式0和方式1的逻辑结构示意图。

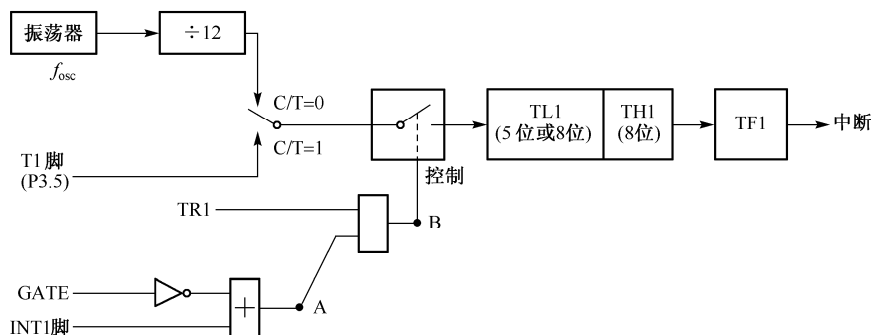


图4.18 定时器/计数器T1方式0和方式1的逻辑结构示意图

图中, TL1在加1计数溢出时向TH1进位, 当TH1加1计数溢出时置“1”中断标志TF1。 $C/\overline{T}=0$ 时, 振荡器的12分频信号($f_{\text{osc}}/12$)作为计数信号, 此时T1用于定时器; $C/\overline{T}=1$ 时, 计数脉冲为T1 (P3.5) 引脚上的外部输入脉冲, 当P3.5发生由高到低的负跳变时, 计数器加1, 这时T1用于外部脉冲计数器, 外部计数脉冲频率不能超过单片机振荡器频率的1/24。

GATE=0时, A点电位常为“1”, B点电位取决于TR1的状态。TR1=1时, B点为高电平, 电子开关闭合, 允许T1计数; TR1=0时, B点为低电平, 电子开关断开, 禁止T1计数。

GATE=1时, A点电位由 $\overline{\text{INT1}}$ (P3.3) 输入电平确定, 仅当 $\overline{\text{INT1}}$ 输入为高电平且TR1=1时, B点才是高电平, 使电子开关闭合, 允许T1计数。

2. 方式2

方式2为自动重装初值的8位定时器/计数器。其逻辑结构如图4.19所示。TL1作为8位计数器, TH1作为常数缓冲器, 当TL1计数器溢出时, 在置“1”溢出中断标志TF1的同时, 将TH1中的初始计数值重新装入TL1, 使其重新开计数。

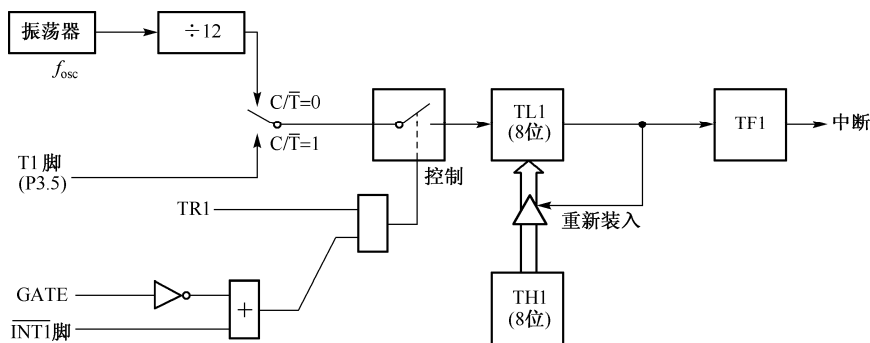


图4.19 定时器/计数器T1方式2的逻辑结构

3. 方式3

方式3只适用于T0, 在一般情况下, 当定时器/计数器T1用于串行口波特率发生器时, 定时器T0才定义为方式3, 将T0分为两个独立的8位计数器TL0和TH0。当T0定义为方式3时, T1仍可定义为方式0、方式1和方式2。

T0在方式3下的逻辑结构如图4.20所示。TL0使用状态控制位 C/\overline{T} 、GATE、TR0、 $\overline{\text{INT0}}$, 而TH0被固定为一个8位定时器（此时不能用于外部计数方式），并使用定时器/计数器T1的状态控制位TR1和TF1, 同时占用T1的中断源。

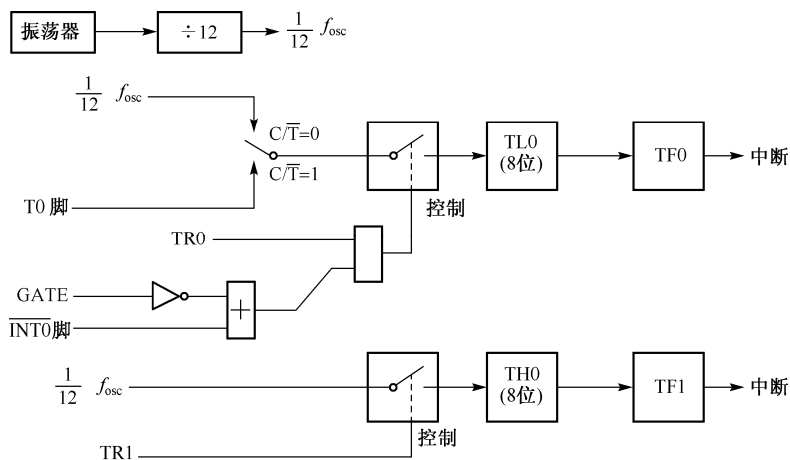


图4.20 定时器/计数器T0方式3的逻辑结构

定时器/计数器T1没有工作方式3，若将T1设置为方式3，将导致T1立即停止计数，其作用相当于使TR1=0。

4.4.2 定时器方式应用编程

8051单片机的定时器/计数器是可编程的，在进行定时或计数操作之前要进行初始化编程。通常8051单片机定时器/计数器的初始化编程包括以下几个步骤：

- 确定工作方式，即给方式控制寄存器TMOD写入控制字；
- 计算定时器/计数器初值，并将初值写入寄存器TL和TH；
- 根据需要对中断控制寄存器IE置初值，决定是否开放定时器中断；
- 使运行控制寄存器TCON中的TR0或TR1置“1”，启动定时器/计数器。

在初始化过程中，要设置定时或计数的初始值时需要进行计算。由于计数器是加法计数，并在溢出时产生中断，因此初始值不能是所需要的计数模值，而是要从最大计数值减去计数模值所得才是应当设置的计数初始值。假设计数器的最大计数值为 M （根据不同工作方式， M 可以是 2^{13} 、 2^{16} 或 2^8 ），则计算初值 X 的公式为

$$\text{计数方式: } X = M - \text{要求的计数值} \quad (4-1)$$

$$\text{定时方式: } X = M - \frac{\text{要求的定时值}}{12 / f_{\text{osc}}} \quad (4-2)$$

例4-8 定时器初值计算。

假设单片机的晶振频率 $f_{\text{osc}} = 6\text{MHz}$ ，现要求产生1ms的定时，试分别计算定时器T1在方式0、方式1和方式2时的初值。

方式0：最大计数值为 $M=2^{13}$ ，因此定时器的初值应为

$$\begin{aligned} X &= 2^{13} - (1 \times 10^{-3}) / (12 / (6 \times 10^{-6})) \\ &= 7692\text{D} \\ &= 1111000001100\text{B} \end{aligned}$$

其中高8位为TH1的初值，即F0H，低5位为TL1的初值。注意，这里TL1的初值应为00001100B，即0CH，而不是60H，因为在方式0时，TL1的高3位是不用的，应都设为0。

方式1：最大计数值为 $M=2^{16}$ ，因此定时器的初值应为

$$\begin{aligned} X &= 2^{16} - (1 \times 10^{-3}) / (2 \times 10^{-6}) \\ &= 65036\text{D} \\ &= 1111111000001100\text{B} \\ &= \text{FE0CH} \end{aligned}$$

此时高8位TH1的初值为FEH，低8位TL1的初值为0CH。

方式2：最大计数值为 $M=2^8$ ，因此定时器的初值应为

$$\begin{aligned} X &= 2^8 - (1 \times 10^{-3}) / (2 \times 10^{-6}) \\ &= 256 - 500 \\ &= -254 \end{aligned}$$

计算得到的初值为负值，说明当 $f_{\text{osc}}=6\text{MHz}$ 时，不能采用方式2（即常数自动装入）来产生1ms的定时，除非把单片机的晶体振荡器频率降得很低。

例4-9 最大定时时间计算。

假设单片机的晶振频率 $f_{osc}=6\text{MHz}$ ，试计算T0在方式0和方式1下的最大定时时间。

T0最大定时时间对应于加法计数器TH0和TL0的各位全为1，即TH0=0FFH，TL0=0FFH，若定时器T0工作在方式0，则最大定时值为

$$T_{\max}=2^{13}\times 12/(6\times 10^6\text{ Hz})=16.384\text{ms}$$

若工作在方式1，则最大定时值为

$$T_{\max}=2^{16}\times 12/(6\times 10^6\text{ Hz})=131.072\text{ms}$$

若要增大定时值，则可以采用降低单片机晶振频率的方法，但这会降低单片机的运行速度，而且定时误差也会加大，故不是最好的方法，而采用软件、硬件相结合的方法则效果较高。

例4-10 用中断方式来扩展定时时间。

设8051单片机的晶振频率为6MHz，利用T0中断扩展方式产生1s定时，当1s定时时间到，从P1.0输出一个低电平点亮发光二极管。Proteus仿真电路如图4.21所示。

本例可以选用方式1，每隔100ms中断一次，中断10次即为1s。定时初值计算如下：

$$\begin{aligned} X &= 2^{16} - (100\times 10^{-3})/(2\times 10^{-6}) \\ &= 15536\text{D} \\ &= 3\text{CB}0\text{H} \end{aligned}$$

因此，TH0 = 3CH，TL0 = B0H。

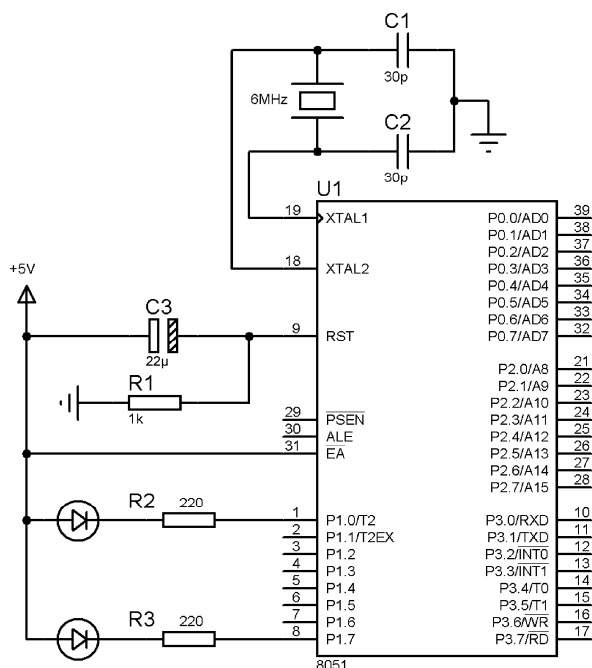


图4.21 利用T0中断扩展方式产生1s定时的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
```



```
#define uint unsigned int

uchar i=10;      //定义中断次数
sbit L1=P1^0;    //定义LED
sbit L2=P1^7;

/***** T0中断服务函数*****/
void t0() interrupt 1 using 1{
    TH0=0x3c;TL0=0xb0;      //重装T0初值
    if(i--!=0)L2=~L2;
    else {
        L1=0;TR0=0;
    }
}

/***** 主函数 *****/
void main(){
    TMOD=0x01;TH0=0x3c;TL0=0xb0; //设置T0工作方式,装入T0初值
    IE=0x82; TR0=1;              //开中断,启动T0
    while(1);                    //等待中断
}
```

例4-11 采用T0中断扩展实现单片机实时时钟。

设8051单片机的晶振频率为6MHz，将内存单元30H、31H、32H分别作为时、分、秒单元，每当定时1秒到时，秒单元内容加1，同时秒指示灯闪；满60秒，则分单元加1，同时分指示灯闪；满60分钟，则时单元加1，同时，时指示灯闪；满24小时后，将时单元清“0”，同时熄灭所有指示灯。Proteus仿真电路如图4.22所示。

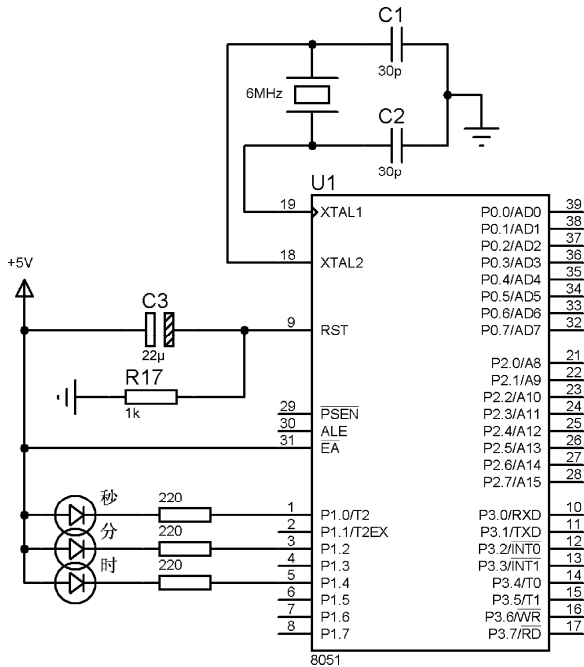


图4.22 利用T0中断扩展方式实现实时时钟的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int
#define SECOND 10

uchar count=0;
sbit L1=P1^0;           //定义LED
sbit L2=P1^2;
sbit L3=P1^4;

struct time {           //定义时、分、秒结构变量
    uchar  hour;        //时
    uchar  min;         //分
    uchar  sec;         //秒
};
struct time clocktime _at_ 0x30; //当前时间

/***** T0中断服务函数 *****/
timer0() interrupt 1 using 2{
    TH0=0x3c; TL0=0xb0;           //重装T0初值
    if( ++count == SECOND ) {     //每中断10次为1秒
        count = 0; L1=~L1;
        if( ++clocktime.sec == 60 ) { //60秒为1分
            clocktime.sec = 0; L2=~L2;
            if( ++clocktime.min == 60 ) { //60分为1小时
                clocktime.min = 0; L3=~L3;
                if( ++clocktime.hour == 24 ) { //24小时为1天
                    clocktime.hour = 0; P1=0x00;
                }
            }
        }
    }
}

/*****主函数*****/
void main(){
    TMOD=0x01; TH0=0x3c; TL0=0xb0; //设置T0工作方式,装入T0初值
    IE=0x82; TR0=1;                //开中断,启动T0
    while(1);                       //等待中断
}
```

例4-12 设8051单片机的晶振频率为6MHz，编写利用T0定时中断在P1.0引脚上产生周期为4ms方波的程序。Proteus仿真电路如图4.23所示，在P1.0引脚上接一个模拟示波器，执行如下程序后，即可看到周期为4ms的方波。

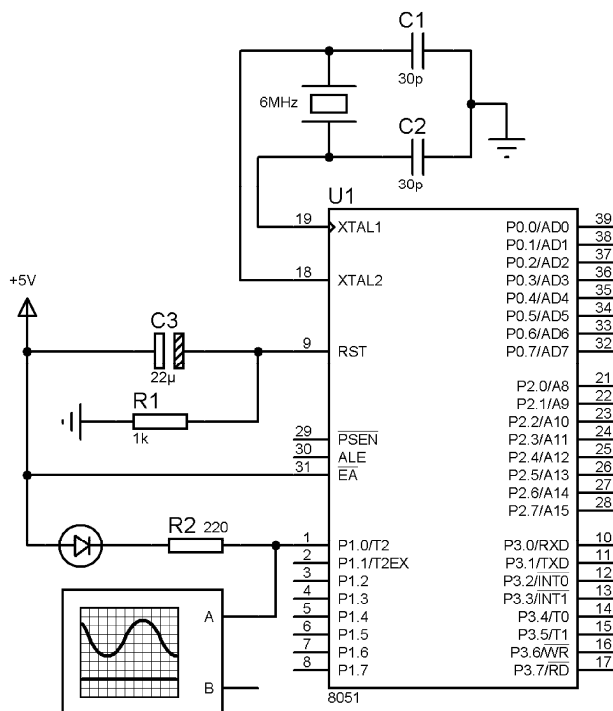


图4.23 利用定时器产生方波的Proteus仿真电路

C51源程序清单如下：

```
#include<reg52.h>

sbit L1=P1^0; //定义LED

/***** T0中断服务函数 *****/
timer0() interrupt 1 using 2{
    TH0=0xfc;TL0=0x18;           //重装T0初值
    L1=~L1;
}

/***** 主函数 *****/
void main(){
    TMOD=0x01;TH0=0xfc;TL0=0x18; //设置T0工作方式，装入T0初值
    IE=0x82; TR0=1;              //开中断，启动T0
    while(1);                    //等待中断
}
```

例4-13 测量脉冲宽度。当特殊功能寄存器TMOD和TCON中的GATE=1、TR1=1，且只有INT1引脚上出现高电平的时候，T1才被允许计数，利用这一特点可以测量加在P3.3（即INT1引脚）上的正脉冲宽度。测量时，先将T1设置为定时方式，GATE设为1，并在INT1引脚为“0”时将TR1置“1”，这样当INT1引脚变为“1”时将启动T1；当INT1引脚再次变为“0”时将停止T1，此时T1的定时值就是被测正脉冲的宽度。若将定时初值设为0，则当单片

机晶振频率为12MHz时，能测量的最大脉冲宽度为65.536ms。

Proteus仿真电路如图4.24 (a) 所示。执行如下程序后暂停，单击“Debug”下拉菜单“8051 CPU Internal (IDATA) Memory”选项，可以看到片内RAM单元30H和31H中内容随外加脉冲宽度而变化，如图4.24 (b) 所示。

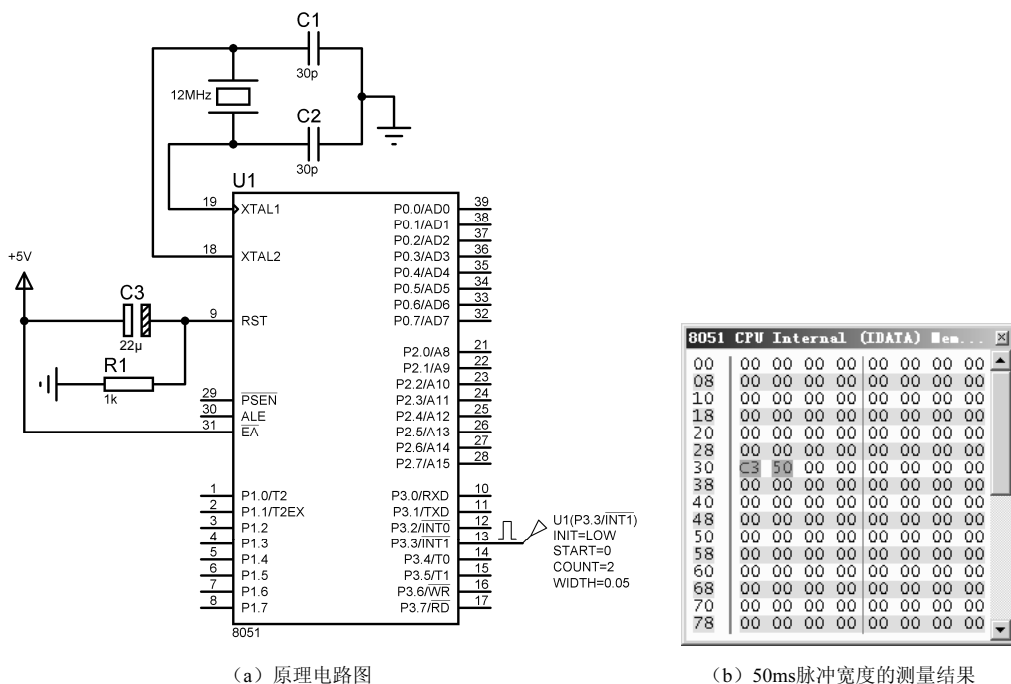


图4.24 测量脉冲宽度的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char

uchar Me[2] _at_ 0x30;
sbit Mp=P3^3; //定义脉冲输入端

/***** 主函数 *****/
void main() {
    TMOD=0x90; TH1=0x00; TL1=0x00; //设置T1工作方式，装入T1初值
    while(Mp); //等待P3.3变低
    TR1=1; //启动T1
    while(!Mp); //等待P3.3变高
    while(Mp); //等待P3.3再次变低
    TR1=0; //停止T1
    Me[0]=TH1; //读取脉冲宽度值；分别存放于30H和31H中
    Me[1]=TL1;
    while(1);
}
```

4.4.3 计数器方式应用编程

采用计数器方式工作时，外部计数脉冲从T0或T1引脚输入，计数脉冲的最高计数频率为单片机晶振频率的1/24，同时还要求计数脉冲的高、低电平保持时间均大于一个机器周期，外部脉冲的下降沿触发计数，当加法计数器累加到工作方式确定的最大计数值时，再来一个外部脉冲将导致计数器溢出。

例4-14 将T0设置为外部脉冲计数方式，在P3.4（T0）引脚上外接一个按钮，每按一次按钮产生一个单脉冲，T0计数一个脉冲，同时将计数值送往P1口外接的LED发光二极管显示。Proteus仿真电路如图4.25所示。

C51源程序清单如下。

```
#include<reg52.h>

/***** 主函数 *****/
void main() {
    TMOD=0x05; TH0=0x00; TL0=0x00; //设置T0工作方式，装入T0初值
    TR0=1;                          //启动T0，开始计数
    while(1) {
        P1=TL0;                      //将记数结果送P1口
    }
}
```

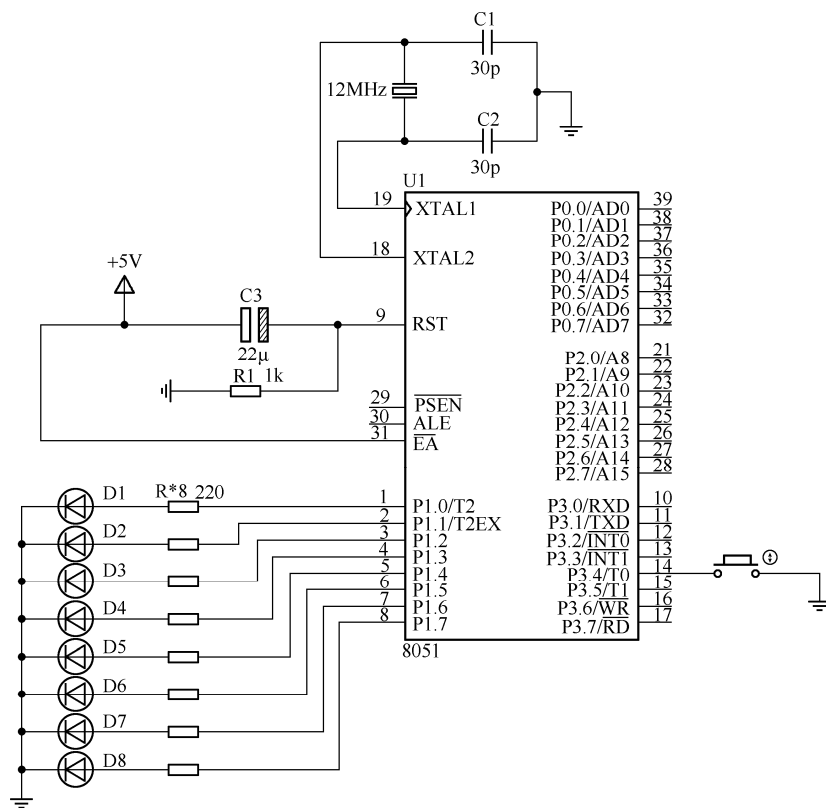


图4.25 T0作为外部计数器应用的Proteus仿真电路

例4-15 要求当P3.4 (T0) 引脚上的电平发生负跳变时, 从P1.0输出一个500μs的同步脉冲。将T0设置为方式2, 外部计数方式, 计数初值设为FFH, 当P3.4引脚上的电平发生负跳变时, T0计数器加1, 同时T1发生溢出使TF0标志置位; 然后将T0改变为500μs定时工作方式, 并使P1.0输出由1变为0。当T0定时时间到, 产生溢出, 使P1.0恢复输出高电平, 同时T0恢复外部计数工作方式。

Proteus仿真电路如图4.26所示, 将P1.0和P3.4引脚分别接到模拟示波器的A、B输入端, 每次按下按钮时, 可以看到P1.0输出的同步脉冲信号。

若单片机晶振频率为6 MHz, 则T0的定时初值应为

$$\begin{aligned} X &= 2^8 - (500 \times 10^{-6}) / (2 \times 10^{-6}) \\ &= 6D \\ &= 06H \end{aligned}$$

C51源程序清单如下。

```
#include<reg52.h>
sbit L=P1^0;

/***** 主函数 *****/
void main() {
    while(1) {
        TMOD=0x06; TH0=0xff; TL0=0xff; //设置T0为8位计数方式, 装入初值
        TR0=1; //启动T0, 开始计数
        while(!TF0); //查询T0溢出标志
        TF0=0; TR0=0; //停止计数
        TMOD=0x02; TH0=0x06; TL0=0x06; //改变T0为8位定时方式, 装入初值
        L=0; TR0=1; //P1.0输出低电平, 启动T0定时500μs
        while(!TF0); //查询T0溢出标志
        TF0=0; L=1; TR0=1; //P1.0输出高电平, 停止T0
    }
}
```

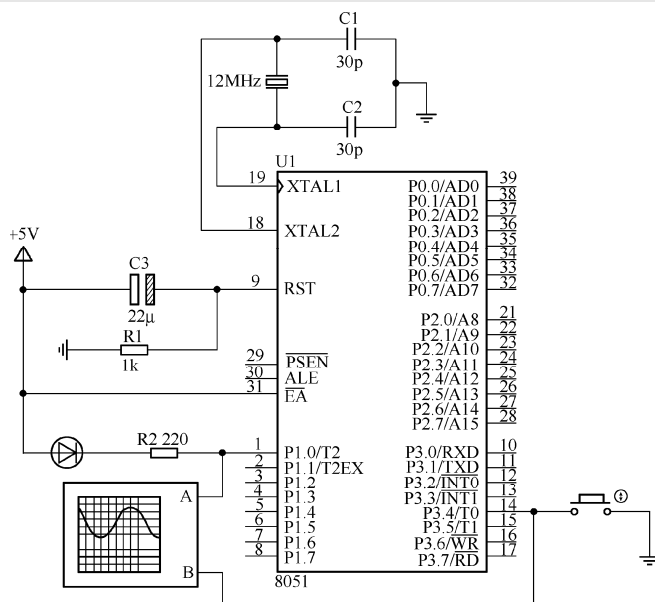


图4.26 产生同步脉冲的Proteus仿真电路

4.4.4 利用定时器产生音乐

声音的频谱范围约在几十到几千赫兹，利用单片机定时器的定时中断功能，可以从一个I/O口线上形成一定频率的脉冲，经过滤波和功率放大，接上喇叭就能发出一定频率的声音。若再利用延时程序控制输出脉冲的频率来改变音调，即可实现音乐发生器功能。

要让单片机产生音频脉冲，必须计算出某一音频的周期，再将此周期除以2得到半周期，利用定时器对此半周期进行定时，每当定时时间到，将某个I/O口线上的电平取反，从而在I/O口线上得到所需要的音频脉冲。产生音频的定时器初值计算公式为

$$t = 2^k - \frac{f_{\text{osc}} / 12}{2 \times F_r} \quad (4-3)$$

式中， k 的值根据单片机工作方式确定，可为13（方式0）、16（方式1）、8（方式2）； f_{osc} 为单片机晶振频率； F_r 为希望产生的音频。

例如，中音DO的频率为523Hz，若单片机晶振频率为12MHz，则定时器T0设置为工作方式1，按公式（4-3）计算得到定时器初值为64580；高音DO的频率为1047Hz，计算得到定时器初值为65058。表4.3为单片机晶振频率为12MHz时，C调各音符频率与定时器初值对照表。

表4.3 C调各音符频率与定时器初值对照表（ $f_{\text{osc}} = 12\text{MHz}$ ）

音 符	频率（Hz）	定时器初值 t	音 符	频率（Hz）	定时器初值 t
低1 DO	220	63263	#4 FA#	622	64732
#1 DO#	233	63390	中5 SO	659	64777
低2 RE	247	63512	#5 SO#	698	64820
#2 RE#	262	63628	中6 LA	740	64860
低3 ME	277	63731	#6 LA#	784	64898
低4 FA	294	63835	中7 SI	831	64934
#4 FA#	311	63928	高1 DO	880	64968
低5 SO	330	64021	#1 DO#	932	64994
#5 SO#	349	64103	高2 RE	988	65030
低6 LA	370	64185	#2 RE#	1046	65058
#6 LA#	392	64260	高3 ME	1109	65085
低7 SI	415	64331	高4 FA	1175	65110
中1 DO	440	64400	#4 FA#	1245	65134
#1 DO#	466	64463	高5 SO	1318	65157
中2 RE	494	64524	#5 SO#	1397	65178
#2 RE#	523	64580	高6 LA	1480	65198
中3 ME	554	64633	#6 LA#	1568	65217
中4 FA	587	64684	高7 SI	1661	65235

一段音乐中除音符之外，还需要节拍，可以通过延时方式来产生不同的节拍。如果1拍为0.4s，则1/4拍为0.1s，只要设定延时时间就可以求得节拍时间。例如，一段延时程序DELAY为1/4拍，则1拍只要调用4次DELAY程序，依此类推。表4.4为1/4和1/8节拍的设定。

表4.4 1/4和1/8节拍的设定 ($f_{osc} = 12\text{MHz}$)

1/4节拍		1/8节拍	
曲调值	延时时间 (ms)	曲调值	延时时间 (ms)
4/4	125	4/4	62
3/4	187	3/4	94
2/4	250	2/4	125

编写音乐程序时, 先把乐谱的音符找出, 确定定时器的初值, 再根据节拍确定延时时间。每个音符使用1字节, 字节的高4位存放音符的高低, 低4位存放音符的节拍。将音符对应的定时器初值表放在TABLE1处, 音符节拍码表放在TABLE处。

“生日快乐”乐谱为

$\underline{5} \underline{5} \underline{6} \underline{5} | 1 \ 7 \ - | \underline{5} \underline{5} \underline{6} \underline{5} | 2 \ 1 \ - | \underline{5} \underline{5} \underline{5} \underline{3} | 1 \ 7 \ 6 | \underline{4} \underline{4} \underline{3} \underline{1} | 2 \ 1 \ - |$

按照上述原理可以编写出“生日快乐”乐曲的C51程序, Proteus仿真电路如图4.27所示, 单击“Play”按钮执行程序, 将从音箱中听到“生日快乐”的乐曲。

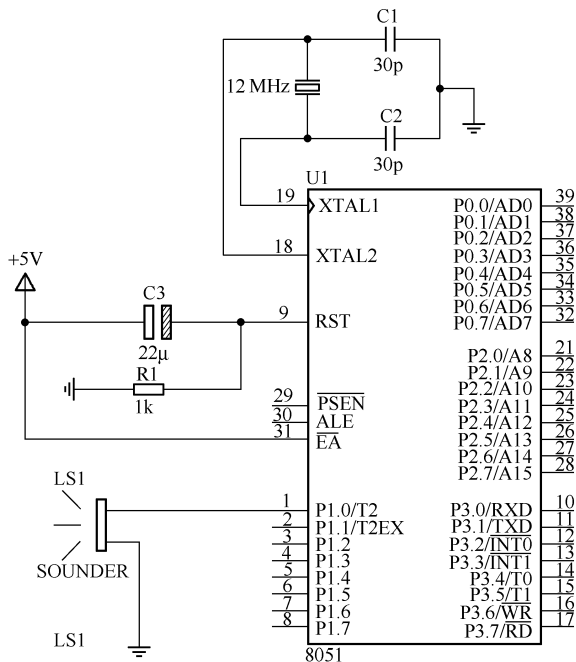


图4.27 利用定时器产生音乐的Proteus仿真电路

例4-16 “生日快乐”C51源程序清单如下。

```
#include<reg52.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int

sbit BEEP=P1^0; //定义喇叭输出端口
uchar tick,t1,th; //定义节拍和T0初值变量
```



```

uchar TABLE[]={ //音符节拍码表
    0x82,0x01,0x81,0x94,0x84,0xB4,0xA4,0x04,
    0x82,0x01,0x81,0x94,0x84,0xC4,0xB4,0x04,
    0x82,0x01,0x81,0xF4,0xD4,0xB4,0xA4,0x94,
    0xE2,0x01,0xE1,0xD4,0xB4,0xC4,0xB4,0x04,
    0x82,0x01,0x81,0x94,0x84,0xB4,0xA4,0x04,
    0x82,0x01,0x81,0x94,0x84,0xC4,0xB4,0x04,
    0x82,0x01,0x81,0xF4,0xD4,0xB4,0xA4,0x94,
    0xE2,0x01,0xE1,0xD4,0xB4,0xC4,0xB4,0x04,
    0x00};

uchar TABLE1[]={ //音符对应的定时器初值表
    0xfb,0x04,0xfb,0x90,0xfc,0x09,0xfc,0x44,
    0xfc,0xac,0xfd,0x09,0xfd,0x34,0xfd,0x82,
    0xfd,0xc8,0xfe,0x06,0xfe,0x22,0xfe,0x56,
    0xfe,0x85,0xfe,0x9a,0xfe,0xc1};

/***** T0中断服务函数 *****/
timer0() interrupt 1 using 1{
    TL0=tl;TH0=th; //重装定时初值
    BEEP=~BEEP; //喇叭输出端口电平取反
}

/***** 基本单位延时函数 *****/
void delay1(){
    uint i;
    for(i=0;i<20000;i++);
}

/***** 节拍延时函数 *****/
void delay(tt){
    uchar i;
    for(i=0;i<=tt;i++) delay1();
}

/***** 主函数 *****/
void main(){
    uchar t,t1,k=0; //定义临时变量
    while(1){
        TMOD=0x01;IE=0x82; //定义T0工作方式,开中断
        while(TABLE[k]!=0){ //判断取得的音符节拍码是否为结束码
            tick=(TABLE[k])&0x0f; //不是,则取节拍码
            t=(_crol_(TABLE[k],4))&0x0f; //取音符码
            if(t!=0){ //判断取得的音符码是否为0
                t1=--t*2+1; //不是,根据取得的音符码计算T0初值
                t=t*2;
                t1=TL0=TABLE1[t1];
            }
        }
    }
}

```

```

        th=TH0=TABLE1[t];
        TR0=1;                                //启动T0
    }
    else TR0=0;                                //取得的音符码为0，则停止T0
    delay(tick);                                //根据则取得的节拍码延时
    k++;
}
TR0=0;                                        //取得结束码，则停止T0
}
}

```

4.5 串行口

8051单片机片内集成了一个全双工的串行口，对外有两根独立的收、发信号线RXD（P3.0）和TXD（P3.1），可以同时接收和发送数据，实现全双工数据传送。串行口可用于串行/并行数据转换，也可用于串行通信。

串行通信采用异步通信方式传送数据，数据在线路上是以一个字（或称字符）为单位进行传送的，通信字符格式如图4.28所示。

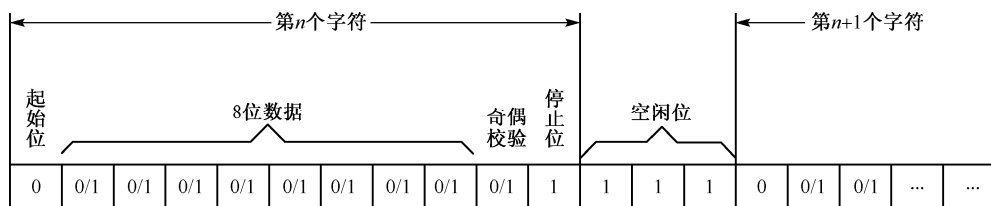


图4.28 异步通信字符格式

一个字符由4个部分组成：起始位、数据位、奇偶校验位和停止位。起始位为“0”，停止位为“1”，线路在不传送数据时保持为“1”。接收端不断检测线路的状态，若连续为“1”以后又检测到一个“0”，就知道又发来了一个新的字符。

起始位后面紧跟的是数据，数据通常为8位，也可以是5位、6位、7位，串行通信速度与数据位数成比例，因此要根据需要来确定数据的位数。奇偶校验位只占一位，也可不用奇偶校验而加一些其他的控制位，如用来确定这个字符所代表信息的性质（是地址还是数据等），这时也可能使用多于1位的附加位。

停止位用来表征字符的结束，它一定是“1”，接收端收到停止位时，就表示一个字符结束，同时也为接收下一个字符做好准备。若停止位以后不是紧接着传送下一个字符，则让线路上保持为“1”。图4.28所表示的是第n个字符与第n+1个字符之间不是紧接着传送的情形，两个字符之间存在空闲位“1”，线路处于等待状态。存在空闲位是异步传送的特征之一。

串行通信有个重要指标叫波特率。它定义为每秒钟传送二进制数码的位数。在异步通信中，波特率为每秒传送的字符数和每个字符位数的乘积。例如，每秒传送的速率为120字符/秒，而每个字符又包含10位（1位起始位，7位数据位，1位奇偶校验位，1位停止位），则波特率为

$$120 \text{ 字符/秒} \times 10 \text{ 位/字符} = 1200 \text{ 位/秒} = 1200 \text{ 波特}$$

进行异步通信时，收、发双方必须事先规定两件事：一是字符格式，即规定字符各部分所占的位数、是否采用奇偶校验及校验方式（偶校验还是奇校验）；二是采用的波特率。

4.5.1 串行口的工作方式与控制

8051单片机与串行口相关的特殊功能寄存器包括数据缓冲器SBUF、串行口控制寄存器SCON和电源控制寄存器PCON。

数据缓冲器SBUF在物理上分为两个独立的发送缓冲器和接收缓冲器。这两个缓冲器占用相同的物理地址99H，它究竟是用于发送缓冲器还是接收缓冲器，取决于软件编程指令。

串行口控制寄存器SCON（地址为98H）包含有串行口的工作方式选择位、接收发送控制位及串行口的状态标志，格式为

D7	D6	D5	D4	D3	D2	D1	D0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0和SM1为串行口的工作方式选择位，详见表4.5。

表4.5 串行口工作方式

SM0	SM1	工作方式
0	0	方式0，移位寄存器方式（用于I/O端口扩展）
0	1	方式1，8位UART，波特率可变（T1溢出率/ n ）
1	0	方式2，9位UART，波特率为 $f_{\text{osc}}/64$ 或 $f_{\text{osc}}/32$
1	1	方式3，9位UART，波特率可变（T1溢出率/ n ）

表中， n 为16或32，取决于特殊功能寄存器PCON中SMOD位的值，SMOD=1时， $n=16$ ；SMOD=0时， $n=32$ 。UART表示通用异步收发器。

8051单片机的串行口有4种工作方式。

① 方式0为移位寄存器输入/输出方式。串行数据从RXD线输入或输出，而TXD线专用于输出时钟脉冲给外部移位寄存器。这种方式主要用于进行I/O端口的扩展，输出时，将片内发送缓冲器中的内容串行地移入到外部的移位寄存器，输入时将外部移位寄存器中的内容移入片内接收缓冲器，波特率固定为 $f_{\text{osc}}/12$ 。

② 方式1为8位异步接收发送。一帧数据有10位，包括1位起始位（0）、8位数据位和1位停止位（1）。串行口电路在发送时能自动插入起始位和停止位，在接收时，停止位进入SCON中的D2位。方式1的传送波特率是可变的，由定时器1的溢出率决定。

③ 方式2为9位异步接收发送。一帧数据包括有11位，除了1位起始位、8位数据位、1位停止位之外，还可以插入第9位数据，字符格式如图4.29所示。

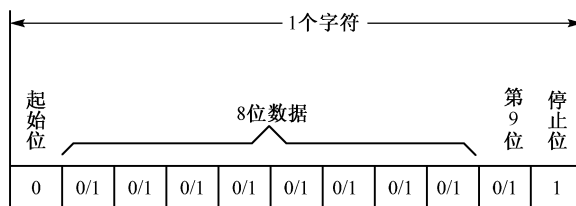


图4.29 串行口方式2的9位UART数据格式

发送时,第9位数据的值可通过SCON中的TB8指定为“0”或“1”,用一些附加的指令可使这一位作为奇偶校验位。接收时,第9位数据进入特殊功能寄存器SCON中的D2位。方式2的波特率为 $f_{\text{osc}}/64$ 或 $f_{\text{osc}}/32$ 。

④ 方式3也是9位异步接收发送,一帧数据有11位,工作方式与方式2相同,只是传送时的波特率受定时器1的控制,即波特率可变。

SCON寄存器中另外各控制位的意义如下:

- SM2为允许在方式2和方式3时进行多机通信的控制位。若允许多机通信,则应使SM2=1,然后根据收到的第9位数据值决定从机是否接收主机的信号,当SM2=0时,禁止多机通信。
- REN为允许串行接收位。由软件置位以允许接收。由软件清“0”来禁止接收。
- TB8为方式2和方式3时发送的第9位数据,需要由软件置位或复位。
- RB8为方式2和方式3时接收到的第9位数据。在方式1,若SM2=0,则RB8是接收到的停止位;在方式0,不使用RB8。
- TI为发送中断标志。由硬件在方式0串行发送第8位结束时置“1”,或在其他方式串行发送停止位的开始时置“1”。该位必须由软件清“0”。
- RI为接收中断标志。由硬件在方式0接收到第8位结束时置“1”,或在其他方式串行接收到停止位的中间时置“1”。该位必须由软件清“0”。

特殊功能寄存器PCON(地址为87H)中,只有一位与串行口工作有关。其格式为

D7	D6	D5	D4	D3	D2	D1	D0
Smod							

串行口工作在方式1、方式2和方式3时,数据传送的波特率与 2^{Smod} 成正比。也就是说,当Smod=1时,将使串行口传送的波特率加倍。

下面对串行口4种工作方式下数据的发送和接收进行稍微详细一点的介绍。

① 方式0: 串行口以方式0工作时,可外接移位寄存器(如74LS164, 74LS165)来扩展I/O端口,也可外接同步输入/输出设备,用同步的方式串行输入或输出数据。在方式0时,串行口相当于一个并入串出(发送)或串入并出(接收)的移位寄存器,数据传送时的波特率是不变的,固定为 $f_{\text{osc}}/12$,数据由RXD(P3.0)端输入,同步移位脉冲由TXD(P3.1)端输出。发送或接收的是8位数据,低位在前。发送或接收完8位数据时,置“1”中断标志TI或RI。

方式0的发送操作是在TI=0的情况下,由一条写发送缓冲器SBUF的指令启动,然后在RXD线上发出8位数据,同时在TXD线上发出同步移位脉冲。8位数据发送完后由硬件置位TI,同时向CPU申请串行发送中断。若中断不开放,则可通过查询TI的状态来确定是否发送完一组数据。当TI=1以后,必须用软件使TI清“0”,再发送下一组数据。

方式0的接收操作是在RI=0的条件下,使REN=1来启动接收过程。接收数据由RXD输入,TXD输出同步移位脉冲。收到8位数据以后,由硬件使RI=1,发出串行口中断申请。RI也必须由软件清“0”,以准备接收下一组数据。

在方式0下,SCON寄存器中的SM2、RB8、TB8都不起什么作用,一般将它们都设置为“0”。

② 方式1: 方式1采用8位异步通信方式,一帧数据有10位,其中起始位和停止位各占1位。方式1的发送也是在发送中断标志TI=0时由一条写发送缓冲器的指令开始的。启动发送

后，串行口能自动地插入一位起始位“0”，在字符结束前插入一位停止位“1”，然后在发送移位脉冲的作用下，依次由TXD线发出数据，一个字符10位数据发送完毕后，自动维持TXD线上的信号为“1”。当8位数据发完，也就是在停止位开始时，使TI置“1”，用以通知CPU可以发送下一个字符。

方式1发送时的定时信号，也就是发送移位脉冲，是由定时器1产生的溢出信号经过16或32分频（取决于Smod的值）而取得的，因此方式1的波特率是可变的。

方式1在接收时，数据从RXD线上输入。当SCON寄存器中的REN置“1”后，接收缓冲器从检测到有效的起始位开始接收一帧数据信息。无信号时，RXD线的状态保持为“1”，当检测到由“1”到“0”的变化时，即认为收到一个字符的起始位，开始接收过程。在接收移位脉冲的控制下，把接收到的数据一位一位地移入接收移位寄存器，直到9位数据（8位有效数据，1位停止位）全部收齐。

在9位数据（8位有效数据，1位停止位）收齐之后，还必须满足以下两个条件，这次接收才真正有效：

- RI=0；
- SM2=0或者接收到的停止位为“1”。

在满足这两个条件时，将接收移位寄存器中的8位数据转存入串行口寄存器SBUF，收到的停止位则进入RB8，并使接收中断标志RI置“1”。若这两个条件不满足，则这一次收到的数据就不装入SBUF，这实际上就相当于丢失了一帧数据，因为串行口马上又开始寻找下一位起始位准备下一帧数据了。事实上这两个有效接收的条件对于方式1来说是很容易满足的。这两个条件真正起作用是在方式2和方式3中。

③ 方式2和方式3：这两种方式都是9位异步接收、发送方式，操作过程完全一样，一帧数据有11位，其中起始位和停止位各占1位。所不同的只是波特率，方式2的波特率只有两种： $f_{osc}/64$ 或 $f_{osc}/32$ ，而方式3的波特率是可以由用户设定的。下面以方式2为例来说明。

方式2的发送包括9位有效数据，必须在启动发送前把第9位数据装入TB8，这第9位数据起什么作用串行口不做规定，完全由用户来安排，它可以是奇偶验位，也可以是其他控制位。

准备好TB8以后，就可以用一条写SBUF的指令启动发送过程。串行口能自动把TB8取出，并装入到第9位数据的位置，再逐一发送出去。发送完毕，使TI=1。这个过程与方式1是相同的。

方式2的接收与方式1也很相似，不同之处是要接收9位有效数据。在方式1时，是把停止位当作第9位数据来处理的，而在方式2（或方式3）中存在着真正的第9位数据。因此，现在有效接收数据的条件为：

- RI=0；
- SM2=0或接收到的第9位数据为“1”。

第一个条件是提供“接收缓冲器空”的信息，即用户已把SBUF中上次收到的数据读走，可以再次写入。第二个条件则提供了某种机会来控制串行接收，若第9位是一般的奇偶校验位，则可令SM2=0，以保证可靠的接收。若第9位数据参与对接收的控制，则可令SM2=1，然后依据所置的第9位数据来决定接收是否有效。

若这两个条件成立，则接收到的第9位数据进入RB8，而前8位数据进入SBUF以准备让

CPU读取，并且置位RI。若以上条件不成立，则这次接收无效，也不置位RI。

特别需要指出的是，在方式1、方式2和方式3的整个接收过程中，保证REN=1是一个先决条件，只有当REN=1时才能对RXD上的信号进行接收。

8051单片机串行口4种工作方式对应着3种波特率。

对于方式0，波特率是固定的，为单片机振荡频率 f_{osc} 的1/12。

对于方式2，波特率由下式计算，即

$$\text{波特率} = \frac{2^{S_{\text{mod}}}}{64} \times f_{\text{osc}} \quad (4-4)$$

式中， S_{mod} 为PCON寄存器中的D7位， f_{osc} 为单片机的振荡频率。

对于方式1和方式3，波特率由定时器1的溢出率决定，计算公式为

$$\text{波特率} = \frac{2^{S_{\text{mod}}}}{32} \times \frac{f_{\text{osc}}}{12} \left(\frac{1}{2^k - \text{定时器T1初值}} \right) \quad (4-5)$$

式中， S_{mod} 为PCON寄存器中的D7位， f_{osc} 为单片机的振荡频率， k 取决于定时器T1的工作方式，T1在方式0时 $k=13$ ，T1在方式1时 $k=16$ ，T1在方式2和方式3时 $k=8$ 。

4.5.2 串口/并口转换应用编程

串行口工作在方式0时为移位寄存器方式，可用于实现串口/并口转换，数据从RXD端输出，移位脉冲从TXD端输出，波特率固定为单片机工作频率的1/12。

例4-17 在串行口外接一个串入并出8位移位寄存器74LS164，实现串口到并口的转换，Proteus仿真电路如图4.30所示，执行如下程序后将看到LED指示灯轮流点亮。

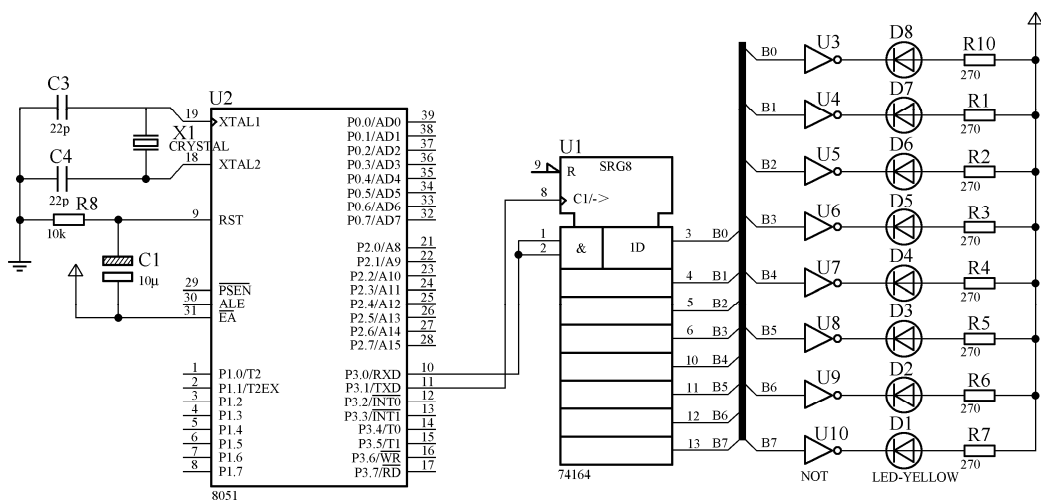


图4.30 利用串行口外接移位寄存器实现串/并转换的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int
```

```

uchar Dat[8]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
/***** 延时函数 *****/
void delay(){
    uint j;
    for(j=0;j<32000;j++);
}

/***** 主函数 *****/
void main(){
    uchar i;
    while(1){
        SCON=0x00;                //设置串行口工作方式0
        for(i=0;i<8;i++){
            SBUF=Dat[i];           //发送数据
            while(!TI);           //检查发送完标志位
            TI=0;
            delay();
        }
    }
}

```

例4-18 在串行口外接一个并入串出8位移位寄存器74LS165，实现并口到串口的转换，Proteus仿真电路如图4.31所示，执行以下程序后，改变拨动开关DIPSWC_8的状态，可以看到LED指示灯会随之变化。

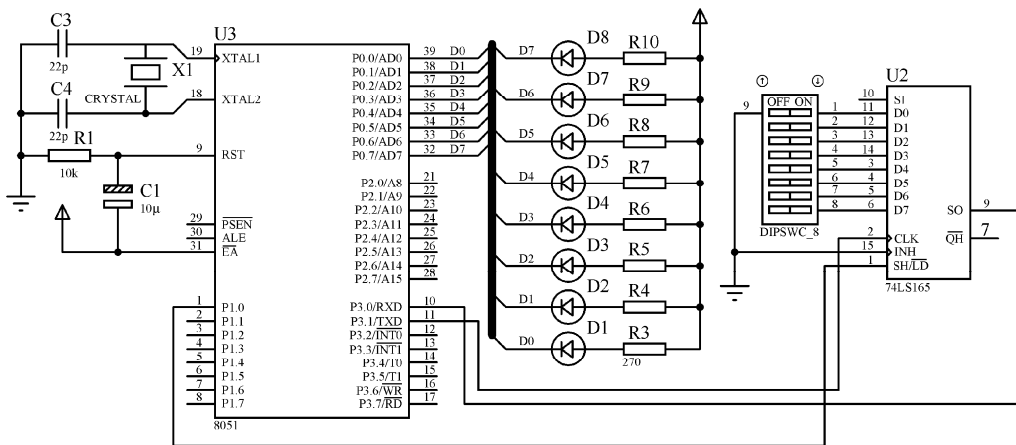


图4.31 利用串行口外接移位寄存器实现并/串转换的Proteus仿真电路

C51源程序清单如下。

```

#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int

sbit shft=P1^0;

```

```

/***** 延时函数 *****/
void delay() {
    uint j;
    for(j=0;j<32000;j++);
}

/***** 主函数 *****/
void main() {
    while(1) {
        shft=0;
        shft=1;
        SCON=0x10;          //设置串行口工作方式0
        while(!RI);         //检查接收标志位
        P0=SBUF;
        RI=0;
        delay();
    }
}

```

4.5.3 单片机与PC机通信应用编程

单片机与PC机之间的通信在实际应用中经常遇到，PC内通常都配有一个RS—232异步串行通信口。其电气性能遵从RS—232标准。RS—232逻辑电平与8051单片机串行口的TTL逻辑电平不兼容，二者必须经电平转换后才能实现正常通信。MAX232芯片是一种常用的电平转换器件。其引脚排列和典型工作电路如图4.32所示。

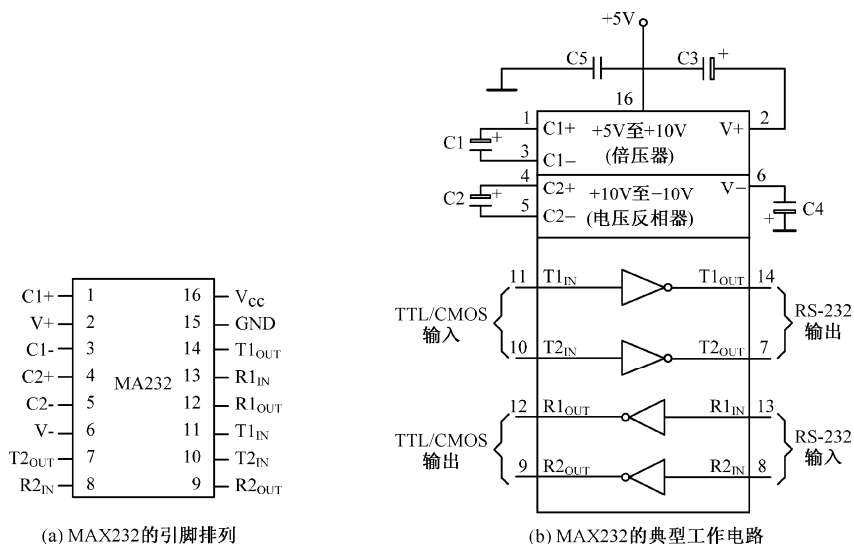


图4.32 MAX232的引脚排列和典型工作电路

下面给出一个利用Proteus软件提供的虚拟终端实现单片机与PC机通信的例子。虚拟终端模拟了PC内部异步串行通信适配器的主要特性，使用十分简单方便。

例4-19 8051单片机与PC之间的串行通信。本例的功能为将PC键盘输入的数据发送给单片机，单片机收到数据后以ASCII码形式从P1口显示接收数据，同时再回送给PC，因此只要PC虚拟终端上显示的字符与键盘输入的字符相同，即说明PC与单片机通信正常。Proteus仿真电路如图4.33所示。

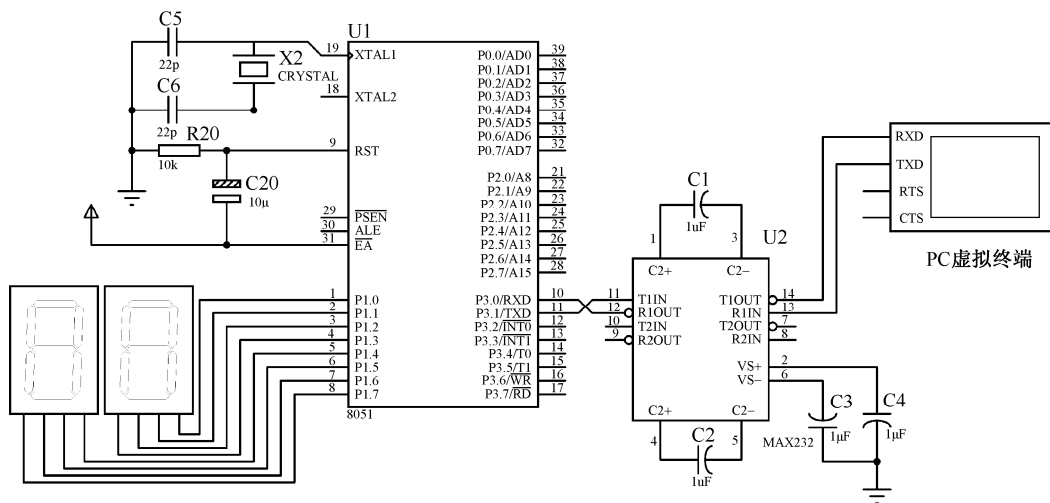


图4.33 8051单片机与PC之间串行通信的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char

/***** 串行口中断服务函数 *****/
void trs() interrupt 4 using 1 {
    uchar Dat1;
    EA=0;
    if(TI==0){                //接收中断
        RI=0;Dat1=SBUF;      //清除中断标志，接收数据
        P1=Dat1;SBUF=Dat1;    //数据从P1口显示，同时回送给PC机
    }
    else TI=0;
    EA=1;
}

/***** 主函数 *****/
void main(){
    SCON=0x50;                //设置串行口工作方式
    TMOD=0x20;                //将T1设为工作方式2
    TH1=TL1=0xf3;PCON=0x80;   //fosc=12MHz时，BD=4800
    TR1=1;                    //启动T1
    ES=1;EA=1;                //开中断
    while(1);                  //等待串行口中断
}
```


发送方C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int
uchar i=0;
uchar Dat[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07, //待发送数据
              0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};

/***** 延时函数 *****/
void delay(){
    uint j;
    for(j=0;j<31000;j++);
}

/***** 主函数 *****/
void main(){
    TMOD=0x20; //将T1设为工作方式2
    TH1=TL1=0xf3;PCON=0x80; //fosc=6MHz时, BD=2400
    TR1=1; //启动T1
    SCON=0xd0; //串行口设为工作方式3, 允许接收
    ES=1;EA=1; //开中断
    ACC=Dat[i];
    CY=P;
    TB8=CY;
    P1=ACC;
    SBUF=ACC; //发送数据
    delay();
    while(1);
}

/***** 发送中断服务函数 *****/
void trs() interrupt 4 using 1 {
    uchar Dat1;
    if(TI==0){ //接收中断
        RI=0;Dat1=SBUF; //清除中断标志, 接收数据
        if(Dat1==0){ //收到回送正确标志
            i++;
            ACC=Dat[i];
            CY=P;
            TB8=CY;
            P1=ACC;
            SBUF=ACC; //启动发送下一个数据
            delay();
            if(i==0x0f) ES=0; //数据发送完毕
        }
    }
    else{
```

```

        ACC=Dat[i]; //收到回送错误标志
        CY=P;
        TB8=CY;
        P1=ACC;
        SBUF=ACC; //重发上一个数据
        delay();
    }
}
else TI=0;
}

```

接收方C51源程序清单如下。

```

#include<reg52.h>
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int
uchar i=0;
uchar Dat[16] _at_ 0x40;

/***** 主函数 *****/
void main(){
    TMOD=0x20; //将T1设为工作方式2
    TH1=TL1=0xf3;PCON=0x80; //fosc=6MHz时, BD=2400
    TR1=1; //启动T1
    SCON=0xd0; //串行口设为工作方式3, 允许接收
    ES=1;EA=1; //开中断
    while(1);
}

/***** 接收中断服务函数 *****/
void res() interrupt 4 using 1 {
    uchar Dat1;
    if(TI==0){ //接收中断
        RI=0;ACC=SBUF;Dat1=ACC; //清除中断标志, 接收数据
        if((P==0&RB8==0)|(P==1&RB8==1)){ //判断奇偶标志
            Dat[i]=Dat1; //奇偶校验正确, 存储数据
            P1=_crol_(Dat1,4);
            i++;
            SBUF=0x00; //回送正确标志
            if(i==0x10) ES=0; //数据接收完毕, 禁止串行口中断
        }
        else{
            SBUF=0xff; //奇偶校验错误, 回送错误标志
        }
    }
    else TI=0;
}

```

在实际应用中经常需要多个微处理器协调工作，由于8051单片机具有多机通信功能，因此利用这一特点很容易组成各种多机系统。典型的主—从多机通信系统如图4.35所示。

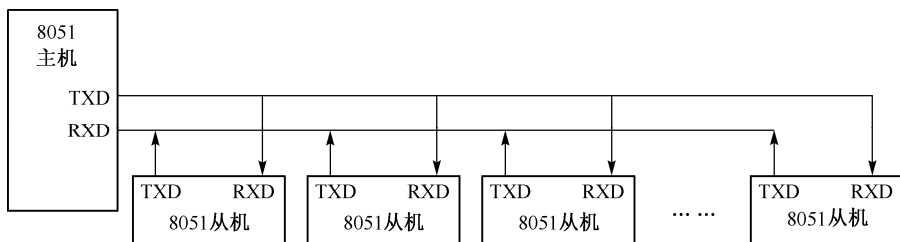


图4.35 典型的主—从多机通信系统

一台8051作为主机，主机的TXD端与其他从机8051的RXD端相连，主机的RXD端与其他从机8051的TXD端相连，主机发送的信息可以被各个从机接收，而各个从机发送的信息只能被主机接收，由主机决定与哪个从机进行通信。

在多机系统中，要保证主机与从机之间可靠的通信，必须要让通信接口具有识别功能，8051单片机串行口控制寄存器SCON中的控制位SM2正是为了满足这一要求而设置的。当串行口以方式2或方式3工作时，发送或接收的每一帧信息都是11位，其中除了包含SBUF寄存器传送的8位数据之外，还包含一个可编程的第9位数据TB8或RB8。主机可以通过对TB8赋予“1”或“0”来区别发送的是地址帧还是数据帧。

根据串行口接收有效条件可知，若从机的SCON控制位SM2为1，则当接收的是地址帧时，接收数据将被装入SBUF并将RI标志置“1”，向CPU发出中断请求；若接收的是数据帧时，则不会产生中断标志，信息将被丢弃。若从机的SCON控制位SM2为0，则无论主机发送的是地址帧还是数据帧，接收数据都会被装入SBUF并置“1”标志位RI，向CPU发出中断请求，因此可以规定如下通信规则：

- 置“1”所有从机的SM2位，使其处于只能接收地址帧的状态；
- 主机发送地址帧，其中包含8位地址信息，第9位为1，进行从机寻址；
- 从机接收到地址帧后，将8位地址信息与其自身地址值相比较，若相同，则清“0”控制位SM2，若不同，则保持控制位SM2为1；
- 主机从第2帧开始发送数据帧，其中第9位为0，对于已经被寻址的从机，因其SM2为0，故可以接收主机发来的数据信息，而对于其他从机，因其SM2为1，将对主机发来的数据信息不予理睬，直到发来一个新的地址帧；
- 若主机需要与其他从机联系，可再次发送地址帧来进行从机寻址，而先前被寻址过的从机在分析出主机发来的地址帧是对其他从机寻址时，恢复其自身的SM2为1，对主机随后发来的数据帧信息不予理睬。

例4-22 本例是一个简单的单片机多机通信系统。一台8051作为主机，另外两台8051作为从机，通信规则如前所述。发往从机1的数据位于主机片内RAM从51H开始的单元中，发往从机2的数据位于主机片内RAM从61H开始的单元中，数据块长度位于50H单元。Proteus仿真电路如图4.36所示。主机端按键K1、K2分别用于设定从机1和从机2地址，按下K1键实现与从机1通信，按下K2键实现与从机2通信。从机将接收到的数据通过P0口显示。

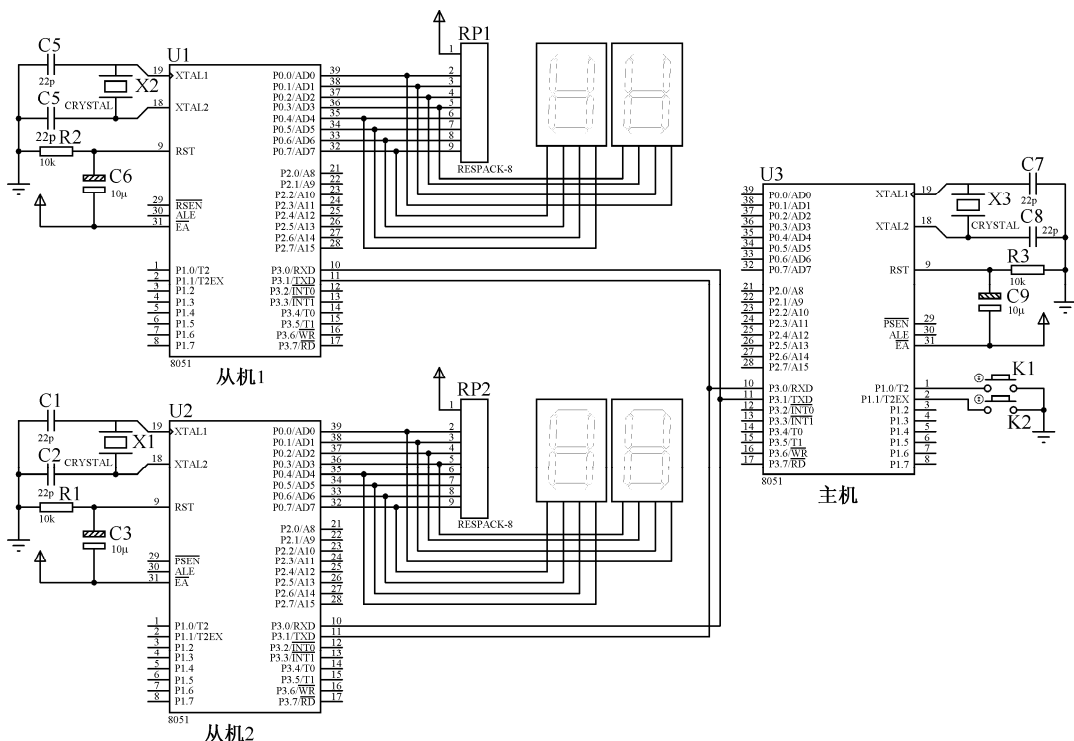


图4.36 主一从方式多机通信系统的Proteus仿真电路

主机采用查询方式发送，每进行一次发送都要判断从机应答，若应答错误，则重发，全部数据发送完毕，最后发送校验和。从机采用中断方式接收，首先接收地址并判断是否与本站地址一致，一致则清“0”从机的SM2控制位，以便继续接收后续数据；否则保持从机的SM2控制位为“1”，放弃接收后续数据。全部数据接收完毕后进行校验和判断，根据校验结果设置接收正确与否的标志。

主机发送C51源程序清单如下。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int

uchar addr,Sum;
uchar bdata flagBase_at_0x20;
uchar Dat1[]={0x01,0x02,0x03};
uchar Dat2[]={0x01,0x02,0x03};
sbit K1=P1^0;
sbit K2=P1^1;
sbit F=flagBase^7;

/***** 数据发送函数 *****/

uchar trs(){
    uchar Dat3,Dat4,DatNum,*ptr;
    SCON=0xd8;          //设置串行口工作方式
    TMOD=0x20;          //将T1设为工作方式2
```

```

    TH1=TL1=0xfd; PCON=0x00; //设置波特率
    TR1=1; //启动T1
    DatNum=0x03; //数据块长度
    do{
        SBUF=addr; //发送从机地址
        while(!TI); //等待发送完
        TI=0;
        while(!RI); //等待从机应答
        RI=0;
        Dat3=SBUF; //接收应答
    }while(Dat3!=0); //应答错误, 重发
    TB8=0;
    do{
        SBUF=DatNum; //发送数据块长度
        while(!TI); //等待发送完
        TI=0;
        while(!RI); //等待从机应答
        RI=0;
        Dat3=SBUF; //接收应答
    }while(Dat3!=0); //应答错误, 重发
    if(F==1) ptr=Dat1;
    if(F==0) ptr=Dat2;
    while(DatNum>0){ //等待发送完
        do{
            Dat4=*ptr; //取发送数据
            SBUF=Dat4;
            while(!TI); //等待发送完
            TI=0;
            while(!RI); //等待从机应答
            RI=0;
            Dat3=SBUF; //接收应答
        }while(Dat3!=0); //应答错误, 重发
        ptr++;
        Sum=Sum+Dat4; //计算数据校验和
        DatNum--;
    }
    SBUF=Sum; //发送校验和
    while(!TI); //等待发送完
    TI=0;
    while(!RI); //等待从机应答
    RI=0;
    Dat3=SBUF; //接收应答
    if(Dat3==0) return 0; //应答正确, 返回0
    else return 1; //应答错误, 返回1
}

```

/****** K1键处理函数 *****/

```

void SET_NM1() { //K1键按下, 设定从机1地址
    addr=0x01; Sum=0x00;

```

```

        F=1;
        Dat1[0]++;Dat1[1]++;Dat1[2]++;
        trs();
        F=0;
    }

    /***** K2键处理函数 *****/
    void SET_NM2() {
        //K2键按下,设定从机2地址
        addr=0x02;Sum=0x00;
        Dat2[0]++;Dat2[1]++;Dat2[2]++;
        trs();
    }

    /***** 主函数 *****/
    void main() {
        while(1) {
            if(K1==0) SET_NM1(); //判断K1键是否按下
            if(K2==0) SET_NM2(); //判断K2键是否按下
        }
    }

```

从机1与从机2接收源程序基本相同,只是本机地址不同。下面仅给出从机1接收数据的C51程序清单。

```

#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int

uchar bdata flagBase _at_ 0x20;
uchar Dat1[3];
uchar j=0;
uchar Sum=0;
uchar DatNum=0;
sbit F=flagBase^7;
sbit F1=flagBase^6;

    /***** 延时函数 *****/
    void delay() {
        uint j;
        for(j=0;j<41000;j++);
    }

    /***** 主函数 *****/
    void main() {
        uchar i;
        SCON=0xf0; //设置串行口工作方式
        TMOD=0x20; //将T1设为工作方式2
        TH1=TL1=0xfd;PCON=0x00; //设置波特率
        TR1=1;ES=1;EA=1; //启动T1,开中断
    }

```



```

F=0;
do{
    //循环显示接收到的数据
    for(i=0;i<0x03;i++){
        P0=Dat1[i];delay();
    }
} while(F==0);
P0=0xff;while(1);
}

/***** 串行口中断服务函数 *****/
void res() interrupt 4 using 1 {
    uchar Dat;
    if(TI==0){
        //接收中断
        RI=0;
        //清除中断标志，接收数据
        if(RB8==1){
            Dat=SBUF;
            //接收从机地址
            if(Dat!=0x01) return;
            //判断是否与本站地址相符
            SM2=0;F1=1; SBUF=0x00;
            return;
        }
        if(F1==1){
            DatNum=SBUF;
            //接收数据块长度
            F1=0;SBUF=0x00;
            return;
        }
        if(DatNum==0){
            Dat=SBUF;
            //接收接收校验和
            if((Dat^Sum)!=0){
                //校验和错误
                SBUF=0xff;F=1;
                return;
            }
            SBUF=0x00;F=0;SM2=1;
            //校验和正确
            Sum=0;j=0;
            return;
        }
        Dat1[j]=SBUF;
        //接收数据
        Sum=Sum+Dat1[j];
        //数据累加
        SBUF=0x00;
        j++;DatNum--;
        return;
    }
    else TI=0;
    return;
}

```

4.5.5 修改底层函数实现printf()重新定向

Keil C51库函数中提供了两个输入、输出函数scanf()和printf()。它们通过底层函数

`_getkey()`和`putchar()`起作用。底层函数默认使用单片机的串行口，用户可以直接应用`scanf()`和`printf()`函数通过串行口实现输入、输出等人机交互功能。

例4-23 利用输入、输出库函数`scanf()`和`printf()`通过串行口实现人机交互功能。

```
#include <reg51.h>          /* 预处理命令 */
#include <stdio.h>
void main() {               /* 主函数 */
    char a;                 /* 主函数的内部变量类型说明 */
    SCON=0x52;              /* 8051单片机串行口初始化 */
    TMOD=0x20;
    TCON=0x69;
    TH1=0x0F3;
    while(1){               /* 利用scanf()和printf()通过串行口实现人机交互 */
        printf ( " \n Please input \n ");
        scanf ("%c", &a);
    }
}                            /* 主程序结束 */
```

用户可以根据需要适当修改底层函数`putchar()`，使其通过其他I/O接口（如液晶显示器LCD接口等）来完成输出功能。例4-24是一个修改`putchar()`函数，实现`printf()`通过LCD显示输出的例子。该例包含两个文件：主程序`main.c`文件和液晶显示`LCD.h`文件。主程序中对Keil C51提供的底层函数`putchar()`进行了修改，通过调用`LCD.h`文件中的功能函数实现光标定位和数据输出，然后就可以直接使用`printf()`函数在LCD上显示输出了。Proteus仿真电路如图4.37所示。

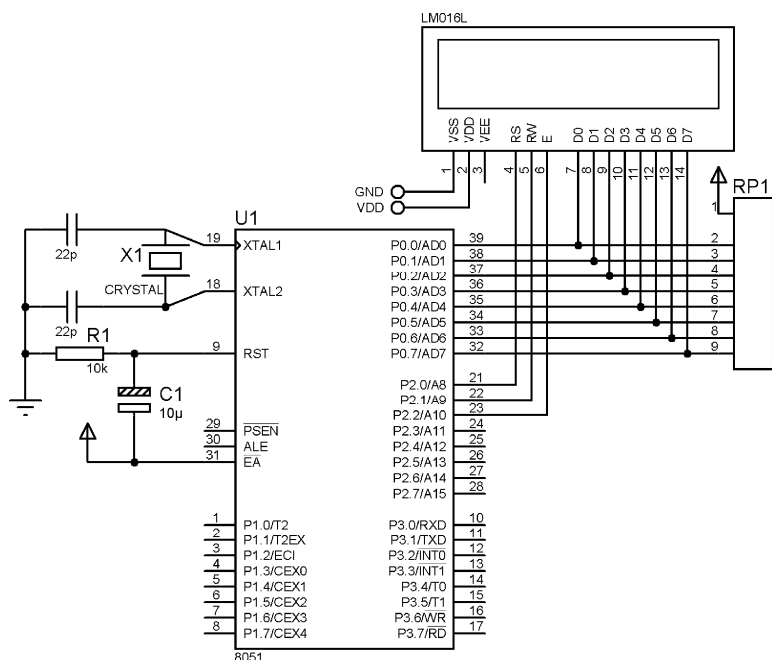


图4.37 单片机使用`printf()`函数在LCD上显示输出的Proteus仿真电路

例4-24 修改底层函数`putchar()`实现`printf()`在LCD上输出。

主程序main.c文件清单。

```
#include <reg51.h>
#include <intrins.h>
#include <stdio.h>
#include <LCD.h>
/***** 主函数 *****/
void main(void){
    LcdInitiate();          /* 调用液晶初始化函数 */
    WriteAddress(0x00);     /* 调用光标定位函数 */
    printf("Hello Everybody\n"); /* LCD显示输出英文字符 */
    printf("Pai = %.4f",3.1415); /* LCD显示输出数字 */
    while(1);
}
/***** 修改底层函数 *****/
char putchar (char c){
    if (c == '\n'){
        WriteAddress(0x40); /* LCD光标定位 */
    }
    else{
        WriteData(c);       /* LCD输出数据 */
    }
    return (c);
}
```

液晶显示LCD.h文件清单。

```
#define uchar unsigned char

sbit RS = P2^0;      // LCD接口引脚定义
sbit RW = P2^1;
sbit E = P2^2;
sbit BF = P0^7;

void LCD_display_data(long num);
char putchar (char c);

/***** 判忙函数 *****/
uchar BusyTest(void){
    bit result;
    RS =0; RW =1; E=1;
    _nop_(); _nop_();
    result = BF;
    E =0;
    return result;
}

/***** 写命令函数 *****/
```

```

void WriteInstruction(uchar dictate){
    while(BusyTest()==1);
    RS =0;  RW =0;  E=0;
    _nop_();  _nop_();
    P0 = dictate;
    _nop_();  _nop_();
    E = 1;
    _nop_();  _nop_();
    E = 0;
}
/***** 指定显示位置函数 *****/
void WriteAddress(uchar x){
    WriteInstruction(x | 0x80);
}
/***** 写数据函数 *****/
void WriteData(uchar y){
    while(BusyTest()==1);
    RS = 1; RW = 0; E = 0;
    P0 = y;
    _nop_();_nop_();
    E = 1;
    _nop_();  _nop_();
    E = 0;
}

/***** 初始化函数 *****/
void LcdInitiate(void){
    WriteInstruction(0x38);
    WriteInstruction(0x0c);    //显示开，无光标，光标不闪烁
    WriteInstruction(0x06);    //光标右移，字符不移动
    WriteInstruction(0x01);    //清屏幕
}

```

系统扩展与低功耗应用

8051单片机用户可以根据不同需要进行系统扩展，包括存储器扩展和输入、输出端口扩展。系统扩展时需要采用地址、数据和控制总线。8051单片机没有单独的外部“三总线”，而是通过以下方法来形成的。

(1) 地址总线

8051单片机的P0口分时输出低8位地址和8位数据信号，P2口输出高8位地址信号。在P0口外面加一个地址锁存器，在地址锁存允许信号ALE的下降沿将低8位地址锁存到锁存器中。低8位地址从P0口外部锁存器输出，而高8位地址直接从P2口输出，形成16根外部地址总线，可寻址 2^{16} 个地址单元，即64KB。

(2) 数据线

从P0口直接输出（不通过外部锁存器）8位数据，形成8根数据总线。

(3) 控制线

单片机的 $\overline{\text{PSEN}}$ 引脚用作程序存储器的输出使能端，P3.6 ($\overline{\text{RD}}$)、P3.7 ($\overline{\text{WR}}$) 分别用作数据存储器的读、写使能端。

5.1 存储器扩展

8051单片机的程序存储器与数据存储器的物理地址空间是相互独立的，可分别扩展64KB的程序存储器和64KB的片外数据存储器。

5.1.1 程序存储器扩展

图5.1为在8051单片机外部扩展一片8K字节程序存储器2764的连接图。图中采用三态输出8D锁存器74LS373作为P0口外部地址锁存器，三态控制端 $\overline{\text{OE}}$ 接地，保证输出常通，锁存控制端G与单片机的ALE端相连，2764的片选端 $\overline{\text{CE}}$ 接地，输出使能端 $\overline{\text{OE}}$ 接单片机的 $\overline{\text{PSEN}}$ 引脚。2764的存储容量为8KB，需要13根地址线。其中，低8位地址线连到单片机P0口外部

锁存器的输出端，高5位地址线直接连到单片机P2口；8位数据线直接连到单片机的P0口；按这种连法2764所占用的地址空间为0000H~1FFFH。

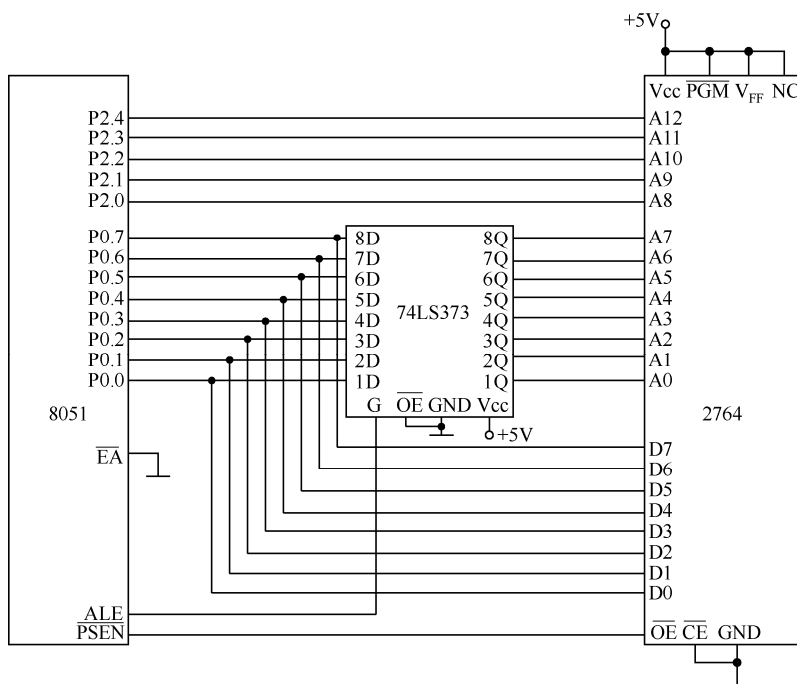


图5.1 在8051单片机外部扩展一片8K字节程序存储器2764的连接图

5.1.2 数据存储器扩展

扩展片外数据存储器时，地址和数据总线连接方法与扩展外部程序存储器相同，但控制总线的连接有所不同：数据存储器的读允许端 \overline{OE} 与单片机的 \overline{RD} 端相连，数据存储器的写允许端 \overline{WE} 与单片机的 \overline{WR} 端相连，单片机ALE端的连接与程序存储器相同。

图5.2为在8051单片机外部扩展一片8K字节数据存储器6264的连接图。图中，8282的功能与74LS373相同。6264的存储容量为8KB，其地址线和数据线与单片机的连接方法与2764相同，其占用的地址空间为0000H~1FFFH。

从8051单片机外部程序存储器和数据存储器的扩展方法可以看到，外部程序存储器的读选通由单片机的 \overline{PSEN} 控制，而数据存储器的读和写选通则由单片机的 \overline{RD} 和 \overline{WR} 控制，因此程序存储器和数据存储器具有相同的逻辑地址空间，但在物理上它们是完全独立的，并且各自具有不同的控制信号，这些信号由执行不同的指令来自动产生，从而可以保证在访问不同存储器地址空间时不会发生混淆。这也是“哈佛式”存储器结构的特点。

例5-1 8051外部扩展8KB数据存储器，采用C51编程将ROM中从0000H地址开始的内容转存到外部RAM中的应用程序。Proteus仿真电路如图5.3所示。执行程序后暂停，通过下拉菜单打开“Memory Contents U3”，可以看到如图5.4所示的存储器内容，已经将单片机片内ROM中的内容传送到片外RAM存储器6264中。

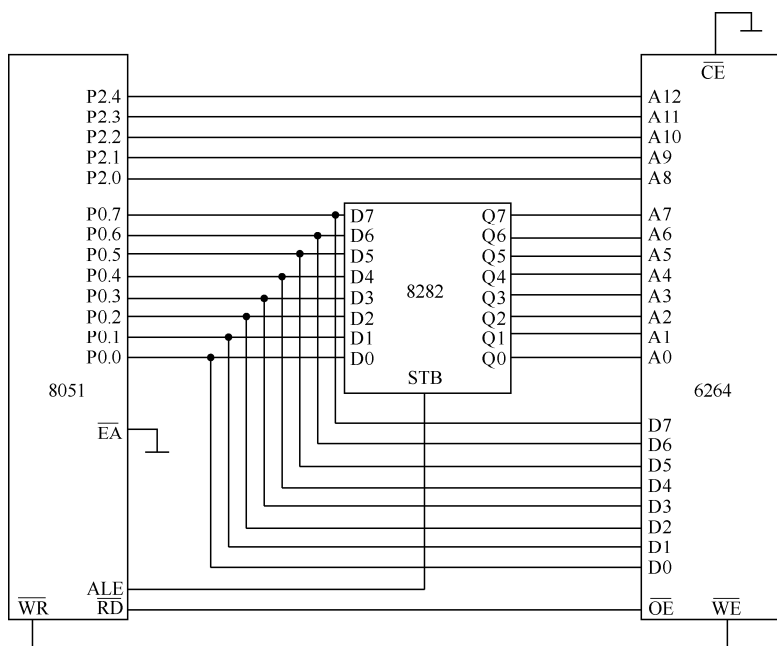


图5.2 在8051单片机外部扩展一片8K字节数据存储器6264的连接图

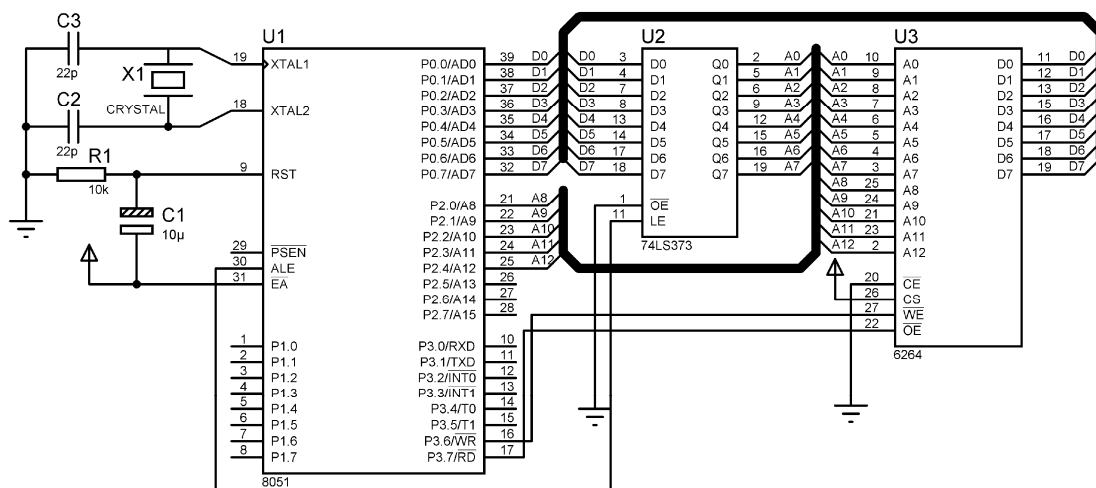
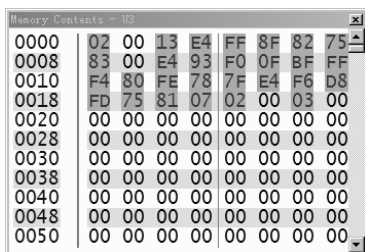


图5.3 在8051单片机外部扩展8KB RAM芯片6264的Proteus仿真电路

C51源程序清单。

```
#include<reg52.h>
#include <absacc.h>
#define uchar unsigned char

void main(){
    uchar i;
    for(i=0;i<0xff;i++)
        XBYTE[i]=CBYTE[i];
    while(1);
}
```



Address	0000	0008	0010	0018	0020	0028	0030	0038	0040	0048	0050
0000	02	00	13	E4	FF	8F	82	75			
0008	83	00	E4	93	F0	0F	BF	FF			
0010	F4	80	FE	78	7F	E4	F6	D8			
0018	FD	75	81	07	02	00	03	00			
0020	00	00	00	00	00	00	00	00			
0028	00	00	00	00	00	00	00	00			
0030	00	00	00	00	00	00	00	00			
0038	00	00	00	00	00	00	00	00			
0040	00	00	00	00	00	00	00	00			
0048	00	00	00	00	00	00	00	00			
0050	00	00	00	00	00	00	00	00			

图5.4 程序执行后6264中的内容

5.2 并行I/O端口扩展

8051单片机提供4个8位的并行I/O端口P0~P3。单片机进行系统扩展时，必须用P0和P2口作为外部地址和数据总线，此时就只有P1口和P3口的一部分口线可以作为并行I/O端口使用。如果应用系统需要较多的I/O端口，就需要进行外部并行I/O端口扩展。

外部并行I/O端口扩展需要占用片外数据存储器地址空间，8051单片机总共16根地址线，寻址范围为64KB。为了唯一地选中某一个外部存储器单元或外部I/O端口，首先要选出该存储器芯片或I/O接口芯片，这称为片选；其次是选出该芯片的某一个存储单元或I/O接口芯片的片内寄存器，这称为字选。常用的选址方法有线选法和地址译码法，分别介绍如下。

5.2.1 线选法

所谓线选法就是利用单片机的一根空闲高位地址线（通常采用P2的某根口线）选中一个外部扩展芯片。选中某个芯片工作时，应对应芯片的片选信号端设为低电平，其他未被选中芯片的片选信号端设为高电平，从而保证只选中指定的芯片工作。在扩展少量外部存储器和I/O端口时采用这种方法。其优点是不需要地址译码器，可以节省器件，降低成本。缺点是可寻址的器件数目受到限制，而且地址空间不连续，这些都会给系统程序设计带来不便。

图5.5为采用线选法进行外部扩展的连接图。在8051外部扩展了一片8KB数据存储器6264、一片可编程接口芯片8255、一片D/A转换芯片0832，分别采用P2.5、P2.6和P2.7作为它们的片选信号。

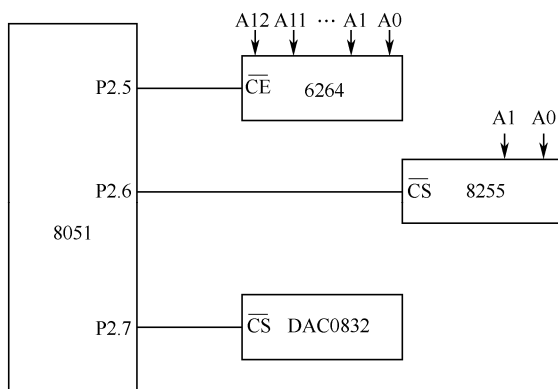


图5.5 采用线选法进行外部扩展的连接图

RAM芯片6264的容量为8KB，需要13根地址线作为字选，因此其片选信号只能用P2.5以上的高位地址线，可以按以下方式计算6264的地址范围。

高8位地址变化范围:	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
	1	1	0	×	×	×	×	×
低8位地址变化范围:	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
	×	×	×	×	×	×	×	×

最高3根地址线的状态固定为110，其余13根地址线的状态为×，表示可以从0变为1，由此可以计算出6264的地址范围为C000H~DFFFH。

8255是一种具有3个I/O端口的可编程接口芯片，除了需要用 \overline{CS} 端作为片选之外，还需要用A1、A0端选择不同的端口。8255的片选端 \overline{CS} 接到8051的P2.6；A0、A1端分别接到最低两位地址线，可以按以下方式计算8255的地址范围。

高8位地址变化范围:	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
	1	0	1	1	1	1	1	1
低8位地址变化范围:	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
	1	1	1	1	1	1	×	×

由此可得8255的地址范围为BFFCH~BFFFH。

D/A转换芯片0832只有一个片选端 \overline{CS} ，当 \overline{CS} 为低电平时选中0832工作。0832的片选端 \overline{CS} 接到8051的P2.7，可以按以下方式计算0832的地址。

高8位地址变化范围:	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
	0	1	1	1	1	1	1	1
低8位地址变化范围:	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
	1	1	1	1	1	1	1	1

由此可得0832的地址为7FFFH。

5.2.2 地址译码法

对于RAM容量较大和I/O端口较多的单片机应用系统进行外部扩展，当芯片所需要的片选信号多于可利用的空闲高位地址线时，就需要采用地址译码法。地址译码法必须采用地址译码器。常用的地址译码器有3-8译码器74LS138、双2-4译码器74LS139等。图5.6为74LS138的引脚排列。表5.1为74LS138的真值表。

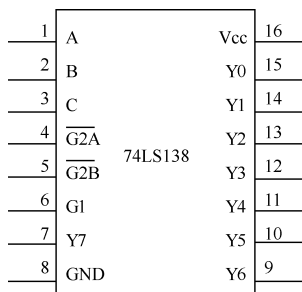


图5.6 74LS138的引脚排列

表5.1 74LS138的真值表

译码器输入						译码器输出
G1	G2A	G2B	C	B	A	Y0~Y7
1	0	0	0	0	0	Y0=0
			0	0	1	Y1=0
			0	1	0	Y2=0
			0	1	1	Y3=0
			1	0	0	Y4=0
			1	0	1	Y5=0
			1	1	0	Y6=0
			1	1	1	Y7=0
0	×	×	×	×	×	Y0~Y7全为1
×	1	×				
×	×	1				

根据表5.1可知，将138译码器的G1接+5V，G2A、G2B接地，将单片机的最高三根地址线P2.7、P2.6、P2.5分别接到138的C、B、A端，利用138的输出端作为外扩芯片的片选端，再将单片机其余13根地址线P2.4~P2.0、P0.7~P0.0作为外扩芯片的字选地址，就可以实现将64KB地址分成8KB×8段，如图5.7（a）所示。此时，138译码器每个输出端的地址范围都是8KB。

在单片机的P2.7引脚上接一个非门，将64KB地址分成32KB×2段，再利用译码器138将32KB地址分成4KB×8段，如图5.7（b）所示，译码器138每个输出端的地址范围都是4KB。

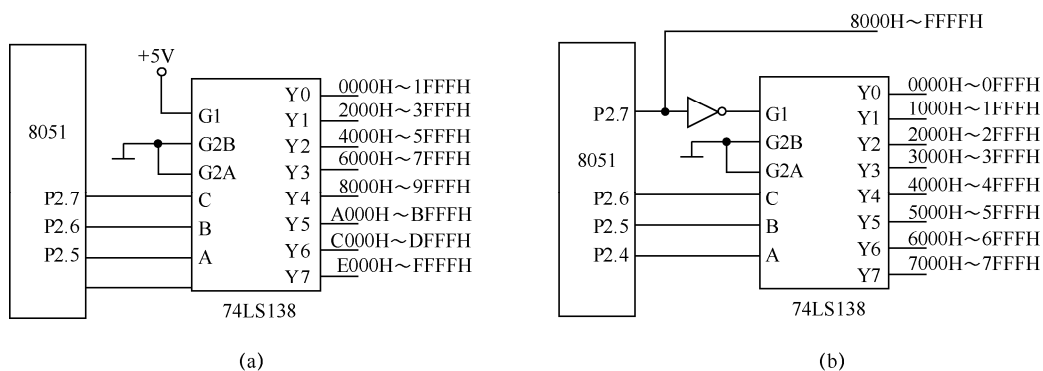


图5.7 用74LS138译码器进行地址空间分配

图5.8为采用地址译码法在单片机外部扩展一片8KB数据存储器6264、一片IO芯片8255、一片定时计数器芯片8253、一片D/A转换器芯片0832的例子。各个外扩芯片的地址编码见表5.2。其中，D/A转换芯片0832的片选端接到138译码器的Y3，地址范围为6000H~7FFFH，通常用最高地址7FFFH作为片选地址，对于8255和8253也是如此。

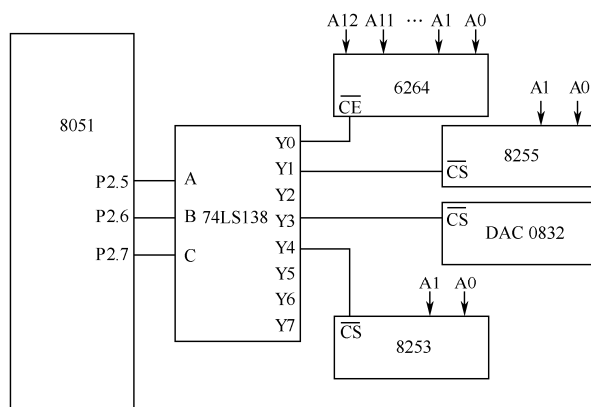


图5.8 采用地址译码法进行外部扩展的连接图

表5.2 图5.8的地址编码

外部器件	片内字节地址数	地址编码
6264	8KB	0000H~1FFFH
8255	4	3FFCH~3FFFH
0832	1	7FFFH
8253	4	9FFCH~9FFFH

5.2.3 8155和8255并行接口扩展芯片应用编程

Intel 8155是一种常用的可编程并行I/O扩展接口芯片。其内部集成有256字节的静态RAM，两个可编程的8位并行端口PA、PB，一个可编程的6位并行接口PC，一个14位的定时计数器。图5.9为Intel 8155的引脚排列和内部逻辑结构。

Intel 8155芯片各引脚功能见表5.3。

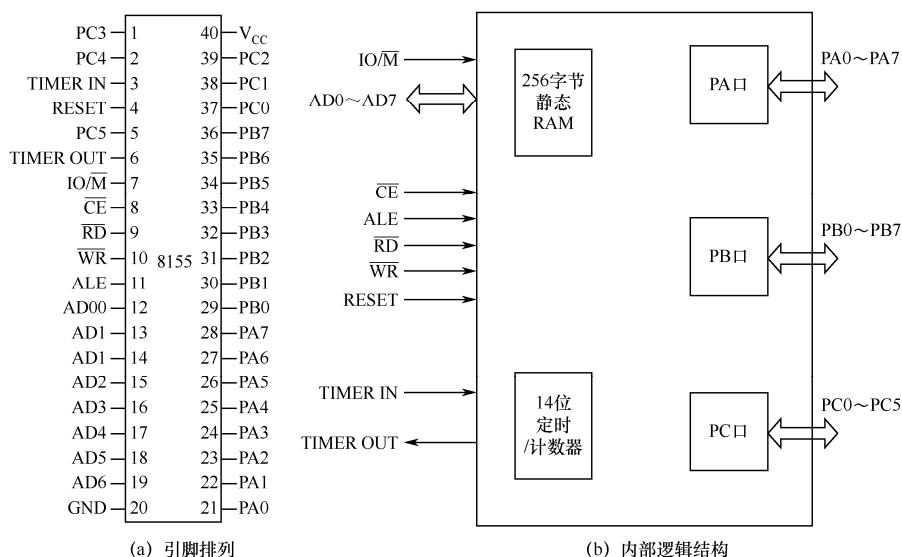


图5.9 Intel 8155的引脚排列和内部逻辑结构

表5.3 Intel 8155芯片各引脚功能

引 脚	功 能
RESET	复位信号，高电平有效。5 μ s左右的正脉冲即可将8155复位，把PA、PB和PC口均初始化为输入方式
AD0~AD7	地址数据线，通常与单片机的P0口相连
ALE	地址锁存信号，通常与单片机的ALE相连
\overline{CE}	片选端，低电平有效
IO/ \overline{M}	用于级联时的串行数据输出端，IO/ \overline{M} =0选中8155片内RAM，此时AD0~AD7输出8155片内RAM地址；IO/ \overline{M} =1选中8155的PA、PB、PC端口、命令/状态寄存器、定时计数器，此时AD0~AD7输出I/O端口地址
\overline{RD}	读选通信号，低电平有效
\overline{WR}	写选通信号，低电平有效
TIMERIIN	定时计数器的输入端
TIMEROUT	定时计数器的输出端
PA0~PA7	PA端口引脚
PB0~PB7	PB端口引脚
PC0~PC5	PC端口引脚
Vcc	正电源端，通常取+5V
GND	模拟地

当Intel 8155芯片的IO/ \overline{M} =1时，其I/O端口地址分配见表5.4。

表5.4 8155的I/O端口地址分配

AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	选中的寄存器
×	×	×	×	×	0	0	0	命令/状态寄存器
×	×	×	×	×	0	0	1	PA口
×	×	×	×	×	0	1	0	PB口
×	×	×	×	×	0	1	1	PC口
×	×	×	×	×	1	0	0	定时计数器的低8位寄存器
×	×	×	×	×	1	0	1	定时计数器的高6位寄存器及工作方式字（2位）

8155内部的命令寄存器和状态寄存器使用同一个端口地址。命令寄存器只能写入不能读出，状态寄存器只能读出不能写入。8155 I/O端口的工作方式由单片机写入命令寄存器的控制字确定，命令字的格式如图5.10所示。

命令字的低4位定义PA、PB和PC口的工作方式。其中，D4、D5位用于设定PA、PB口以选通输入、输出方式工作时是否允许申请中断，D6、D7位为定时计数器的运行控制位。

8155 I/O端口的工作方式如下：

当8155编程为ALT1、ALT2时，PA、PB、PC口均工作在基本输入、输出方式；

当8155编程为ALT3时，PA口定义为选通输入、输出方式，PB口定义为基本输入、输出方式；

当8155编程为ALT4时，PA和PB口均定义为选通输入、输出工作方式。

8155内部状态寄存器用来锁存I/O端口和定时计数器的当前状态，以供CPU查询。状态寄存器只能读出，不能写入，状态寄存器和命令寄存器共用一个口地址。8155状态寄存器的格式如图5.11所示。

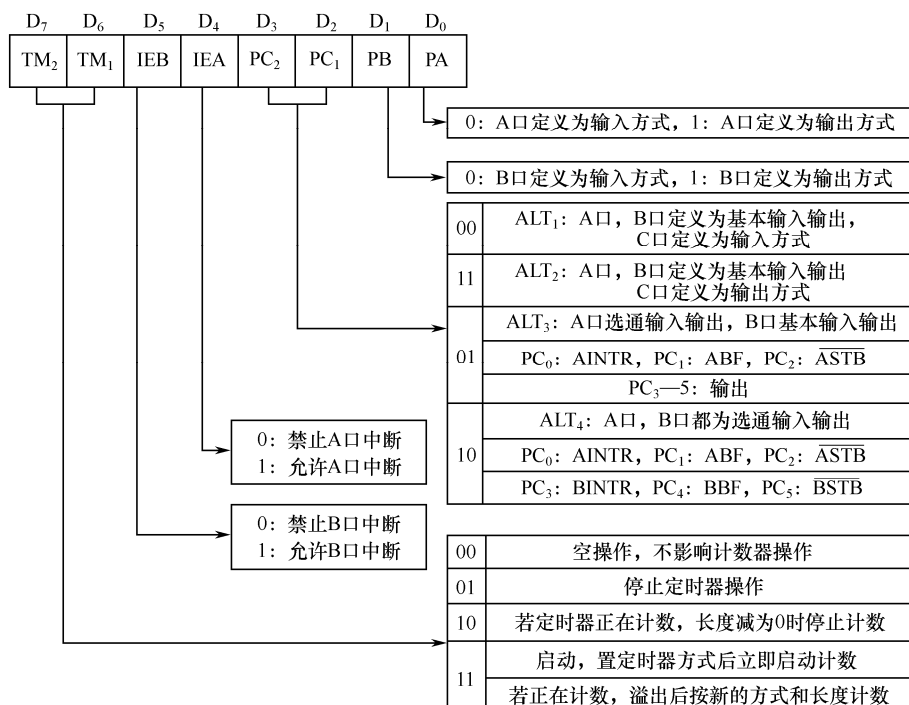


图5.10 8155命令字的格式

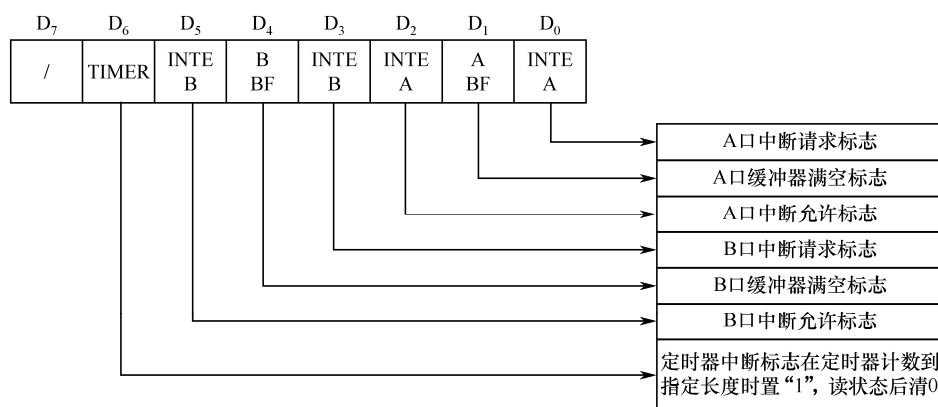


图5.11 8155状态寄存器的格式

8155的片内定时计数器为14位减法计数器, 由两个字节组成。其格式如图5.12所示。它有四种工作方式, 由M2、M1两位确定, 每一种工作方式的输出波形如图5.13所示。

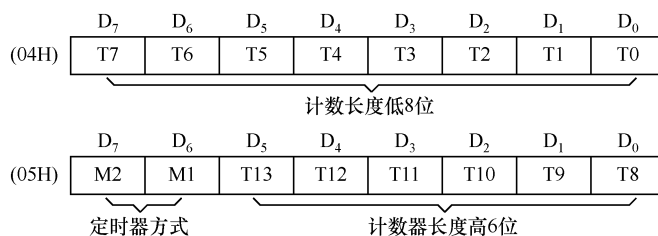


图5.12 8155定时器的格式





M2 M1	方式	定时器输出波形
0 0	单方波	
0 1	连续方波	
1 0	单脉冲	
1 1	连续脉冲	

图5.13 定时方式和输出波形

对定时计数器进行编程时，先要将计数常数和工作方式送入定时计数器口地址（定时计数器低8位、定时计数器高6位、定时器方式M2，M1）。计数常数在0002H~3FFFH之间选择。定时计数器的启动和停止由命令寄存器的最高两位控制。

任何时候都可以设置定时计数器的长度和工作方式，然后必须将启动命令写入命令寄存器中，即使计数器在计数期间，写入启动命令后仍可改变其工作方式。如果写入定时计数器的常数值为奇数，则输出的方波不对称。8155复位后并不预置定时计数器的工作方式和计数常数值。若作为外部事件计数，则由定时计数器状态求取外部输入事件脉冲的方法如下：

停止计数，分别读取定时计数器的两个字节，取低14位计数值，若为偶数，则右移一位即为外部输入事件的脉冲数；若为奇数，则右移一位后再加上计数初值的二分之一的整数部分作为外部输入事件的脉冲数。

8155可以直接与8051单片机接口，不需要任何外部附加逻辑。8155具有片内地址锁存器，可以将单片机的P0口8根引脚直接与8155的AD0~AD7相连。图5.14为8155与8051的基本连接方法。片选信号和 $\text{IO}/\overline{\text{M}}$ 信号分别接到8051的P2.7和P2.0。根据表5.4可知8155的端口地址编码为：

命令/状态寄存器地址：7F00H。

片内RAM字节地址：7E00H~7EFFH。

PA口地址：7F01H。

PB口地址: 7F02H。

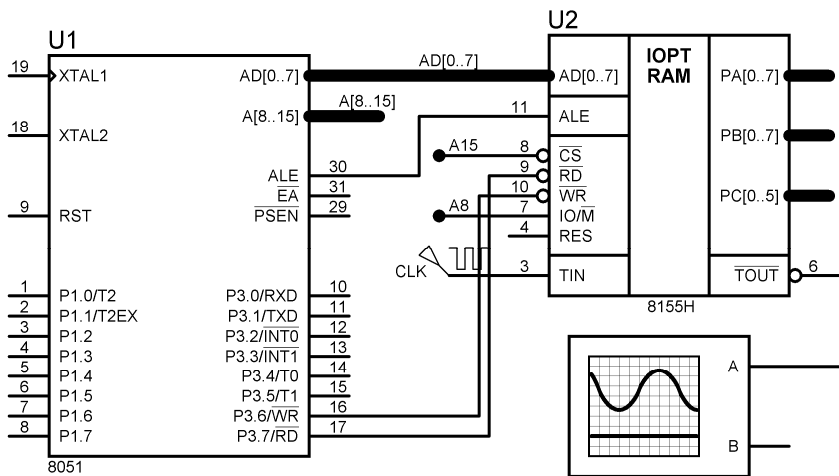


图5.14 8155与8051的基本连接方法

PC口地址：7F03H。

定时计数器低位地址：7F04H。

定时计数器高位地址：7F05H。

例5-2 采用C51编写的8155接口应用程序，将8155的PA口定义为基本输入方式，PB口定义为基本输出方式，定时器对输入脉冲进行15分频并输出连续方波。

```
#include<reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define CADDR 0x7F00 //定义8155命令口地址
#define PORTA 0x7F01 //定义8155PA口地址
#define PORTB 0x7F02 //定义8155PB口地址
#define PORTC 0x7F03 //定义8155PC口地址
#define TIMEL 0x7F04 //定义定时器低位地址
#define TIMEH 0x7F05 //定义定时器高位地址

void main(){
    XBYTE[TIMEL]=0x0f;
    XBYTE[TIMEH]=0x40;
    XBYTE[CADDR]=0xc2;
    while(1);
}
```

Intel 8255也是一种常用并行I/O扩展芯片。图5.15为Intel 8255的引脚排列和内部逻辑结构。它具有3个8位的并行I/O端口，分别称为PA、PB、PC。其中，PA口具有一个8位数据输出锁存/缓冲器和一个8位数据输入锁存器，可编程为8位输入/输出或双向寄存器。PB口与PA口类似，不同的是PB口不能用作双向寄存器。PC与PA口类似，不同的是PC口又分高4位口

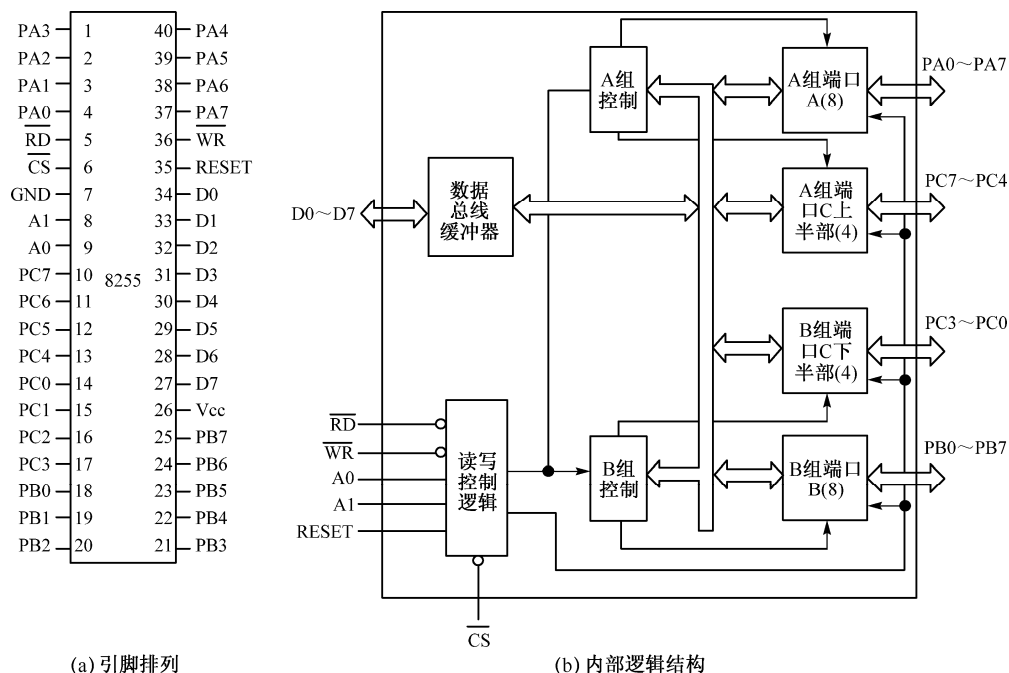


图5.15 Intel 8255的引脚排列和内部逻辑结构

(PC7~PC4) 和低4位口 (PC3~PC0), PC口除了用作输入/输出口之外, 还可用作PA、PB口选通工作方式下的状态控制信号。

Intel 8255芯片各引脚的功能见表5.5。

表5.5 Intel 8255芯片引脚的功能

引 脚	功 能
RESET	复位信号, 高电平有效
\overline{CS}	片选端, 低电平有效
\overline{RD}	读选通信号, 低电平有效
\overline{WR}	写选通信号, 低电平有效
D0~D7	三态双向数据总线, 通过它实现8位数据的读/写操作, 控制字和状态信息也通过数据总线传送
PA0~PA7	PA端口引脚
PB0~PB7	PB端口引脚
PC0~PC7	PC端口引脚
A1、A0	地址选择线, 用来选择8255的PA口、PB口、PC口和控制寄存器
Vcc	正电源端, 通常取+5V
GND	模拟地

8255内部的A组和B组控制电路, 根据CPU的命令控制8255的工作方式, 每组控制电路从读、写控制逻辑接受各种命令, 从内部数据总线接受控制字并发出适当的命令到相应的端口。A组控制电路控制PA口及PC口的高4位, B组控制电路控制PB口及PC口的低4位。8255内部读/写控制逻辑用于管理所有的数据、控制字或状态字的传递, 接受来自CPU的地址及控制信号来控制各个端口的工作状态。其控制信号有:

- 复位信号RESET: 高电平有效。复位时控制寄存器被清“0”, 所有端口都设置为输入方式。
- 片选信号 \overline{CS} : 低电平有效。允许8255与CPU交换信息。
- 读信号 \overline{RD} : 低电平有效。允许CPU从8255端口读取数据或外设状态信息。
- 写信号 \overline{WR} : 低电平有效。允许CPU将数据、控制字写入8255中。
- 端口选择信号A1、A0: 它们与 \overline{RD} 、 \overline{WR} 及 \overline{CS} 信号配合来选择I/O端口及内部控制寄存器, 并控制信息的传送方向, 见表5.6。

表5.6 8255的端口选择及其功能

A1	A0	\overline{RD}	\overline{WR}	\overline{CS}	功 能 说 明
0	0	0	1	0	A口→数据总线
0	1	0	1	0	B口→数据总线
1	0	0	1	0	C口→数据总线
0	0	1	0	0	数据总线→A口
0	1	1	0	0	数据总线→B口
1	0	1	0	0	数据总线→C口
1	1	1	0	0	数据总线→控制寄存器
×	×	×	×	1	数据总线为三态
1	1	0	1	0	非法状态
×	×	1	1	0	数据总线为三态

8255有三种工作方式：方式0、方式1和方式2，如图5.16所示。

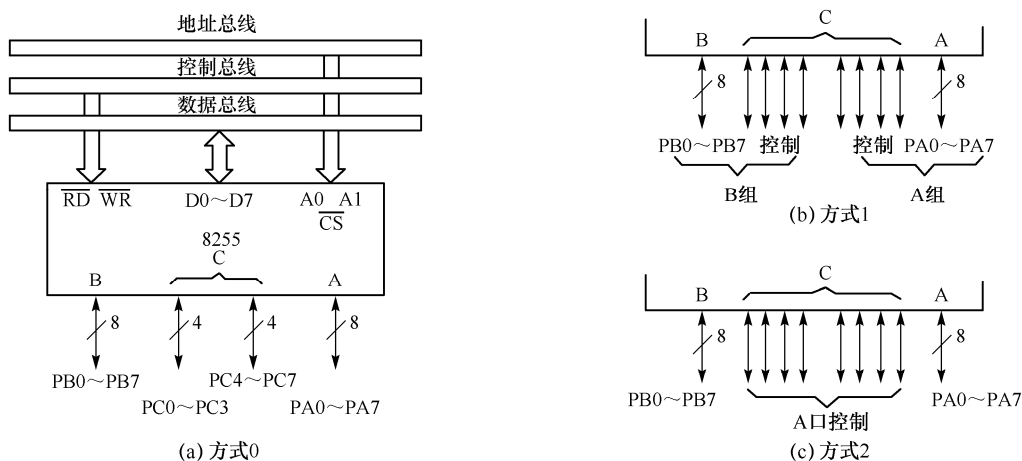


图5.16 8255的三种工作方式

方式0为基本输入/输出方式，PA、PB和PC口都可以设定为输入或输出，作为输出口时，输出的数据被锁存，作为输入口时，输入数据不锁存。

方式1为选通输入/输出方式，PA、PB和PC三个口分为两组：A组包括PA口和PC口的高4位，PA口可编程设定为输入或输出，PC口高4位用作输入、输出操作的控制和同步信号。B组包括PB口和PC口的低4位，PB口可编程设定为输入或输出，PC口低4位用作输入/输出操作的控制和同步信号。PA和PB口的输入/输出数据都被锁存。

方式2为双向总线方式，仅用于PA口，PA为8位双向总线端口，PC的PC3~PC7用作输入、输出同步控制信号，此时PB口只能编程设定为方式0或方式1。

PC口在方式1和方式2时，8255内部规定的联络信号见表5.7所示。

表5.7 PC口的联络信号分布

位	方式 1		方式 2	
	输 入	输 出	输 入	输 出
PC7	I/O	$\overline{\text{OBFA}}$	×	$\overline{\text{OBFA}}$
PC6	I/O	$\overline{\text{ACKA}}$	×	$\overline{\text{ACKA}}$
PC5	IBFA	I/O	IBFA	×
PC4	$\overline{\text{STBA}}$	I/O	$\overline{\text{STBA}}$	×
PC3	INTRA	INTRA	INTRA	INTRA
PC2	$\overline{\text{STBB}}$	$\overline{\text{ACKB}}$	I/O	I/O
PC1	IBFB	$\overline{\text{OBFB}}$	I/O	I/O
PC0	INTRB	INTRB	I/O	I/O

其中，用于输入的联络信号有：

- 选通脉冲输入 $\overline{\text{STBA}}$ 和 $\overline{\text{STBB}}$ ：低电平有效。当外设送来 $\overline{\text{STBA}}$ / $\overline{\text{STBB}}$ 信号时，输入数据被装入8255的锁存器。
- 输入缓冲器信号IBFA和IBFB：高电平有效。表示数据已经装入锁存器，可作为送出的状态信号。

- 中断请求INTR：高电平有效。当 $\overline{IBFA}/\overline{IBFB}=1$ 、 $\overline{STBA}/\overline{STBB}=1$ 时才有效，用来向CPU请求中断服务。

输入操作过程如下：当外设的数据准备好以后，发出 $\overline{STBA}/\overline{STBB}=0$ 信号，输入数据装入8255的锁存器，装满后使 $\overline{IBFA}/\overline{IBFB}=1$ ，CPU可以查询这个状态信息，以决定是否接收8255的数据。或者当 $\overline{STBA}/\overline{STBB}$ 重新变高时，INTR有效，向CPU申请中断，CPU在中断服务程序中接收8255的数据，并使 $\text{INTR}=0$ 。

用于输出的联络信号有：

- 响应信号输入 $\overline{ACKA}/\overline{ACKB}$ ：低电平有效。它是当外设取走8255的数据后发出的响应信号。
- 输出缓冲器信号 \overline{OBFA} 和 \overline{OBFB} ：低电平有效。当CPU把数据送入8255的锁存器后有效，用这个输出的低电平通知外设开始接收数据。
- 中断请求信号INTR：高电平有效。当外设处理完一组数据后， $\overline{ACKA}/\overline{ACKB}$ 变低，并且当 $\overline{OBFA}/\overline{OBFB}$ 变高，然后 $\overline{ACKA}/\overline{ACKB}$ 又变高后使INTR有效，申请中断，进入下一次输出过程。

用户可以通过编程对PC口相应位进行置“1”或清“0”来控制8255的开中断或关中断。8255有两种控制字，即控制PA、PB、PC口工作方式的方式控制字和控制PC口各位置“1”或清“0”的控制字。两种控制字写入的控制寄存器相同，只是用D7位来区分是哪一种控制字：D7=1为工作方式控制字，D7=0为PC口置“1”或清“0”控制字。这两种控制字的格式如图5.17所示。

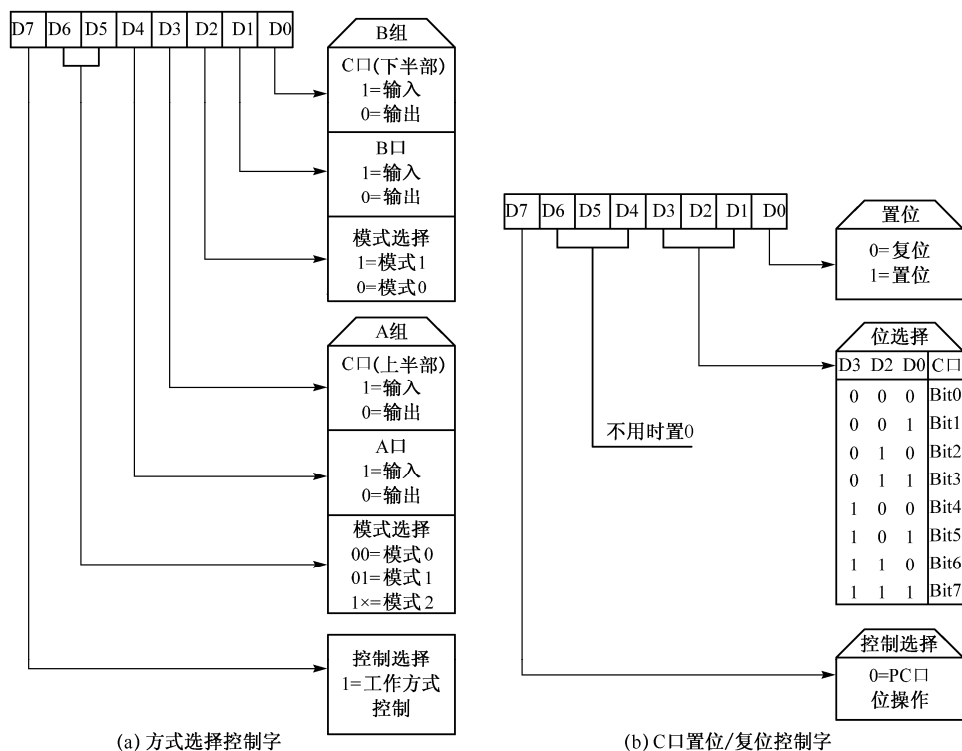


图5.17 8255的控制字格式

例5-3 利用8255与8051单片机接口实现输入、输出的Proteus仿真电路如图5.18所示。图中，8255的片选信号 \overline{CS} 连到8051的P2.7，端口地址选择信号A1、A0由P2.1、P2.0提供。根据表5-6可知，该电路中8255的PA、PB、PC及控制口的地址分别为7CFFH、7DFFH、7EFFH、7FFFH。编程实现8255的PA口按方式0输出，PB口按方式0输入，将PB口外接8个开关的状态通过PA口外接的LED灯反映出来。

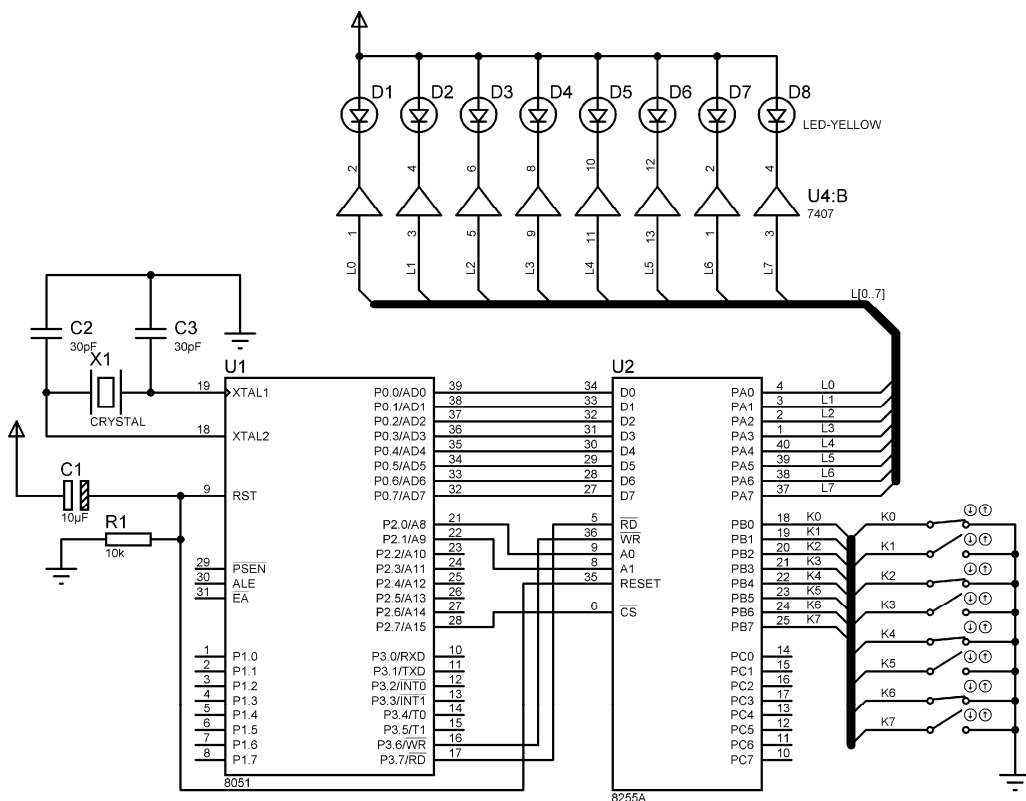


图5.18 8255与8051单片机接口实现输入、输出的Proteus仿真电路

C51源程序清单如下。

```
#include<reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define PORTA  0x7CFF    //定义8255PA口地址
#define PORTB  0x7DFF    //定义8255PB口地址
#define PORTC  0x7EFF    //定义8255PC口地址
#define CADDR  0x7FFF    //定义控制口地址

void main() {
    XBYTE[CADDR]=0x82;
    while(1){
        XBYTE[PORTA]=XBYTE[PORTB];
    }
}
```

5.3 8051单片机的低功耗应用

8051单片机提供了两种低功耗工作方式：空闲方式和掉电方式。这种低功耗工作方式特别适用于采用干电池供电或停电时依靠备用电源供电的单片机应用系统。单片机正常工作时电流为11~20mA，空闲状态时为1.7~5mA，掉电状态时为5~50 μ A。在空闲方式下，振荡器保持工作，时钟脉冲继续输出到中断、串行口、定时器等功能部件，使它们继续工作，但时钟脉冲不再送到CPU，因而CPU停止工作。在掉电方式下，振荡器停止工作，单片机内部所有的功能部件全部停止工作。

单片机的低功耗工作方式是由特殊功能寄存器PCON（地址为87H）控制的，格式为

D7	D6	D5	D4	D3	D2	D1	D0
SMOD	\	\	\	GF1	GF0	PD	IDL

其中各位的意义如下：

- SMOD为串行口的波特率控制位，SMOD=1时波特率加倍；
- GF1、GF0为通用标志位，由用户设定其意义；
- PD为掉电方式控制位，PD置“1”后使器件立即进入掉电方式；
- IDL为空闲方式控制位，IDL置“1”后使器件立即进入空闲方式，若PD和IDL同时置“1”，则使器件进入掉电工作方式。

5.3.1 空闲工作方式

当CPU执行一条置“1”PCON.0（IDL位）的指令后，就进入空闲工作方式。该指令应是CPU执行的最后一条指令。该指令执行完后，CPU即停止工作，进入空闲方式。此时，中断、串行口、定时器还继续工作，堆栈指针SP、程序计数器PC、程序状态字PSW、累加器ACC、片内RAM及其他特殊功能寄存器的内容保持不变，引脚保持进入空闲方式时的状态，ALE和PSEN保持逻辑高电平。

进入空闲方式以后，有两种方法使器件退出空闲方式。一是被允许的中断源请求中断时，由内部的硬件电路清“0”PCON.0位，终止空闲工作方式，CPU响应中断，执行中断服务程序，中断处理完以后，从激活空闲方式指令的下一条指令开始执行程序。

PCON寄存器中的GF0和GF1可用来指示中断是发生在正常工作状态，还是发生在空闲工作状态。CPU在置“1”PCON.0位激活空闲方式的同时，可以先置“1”标志位GF0或GF1，由于产生了中断而退出空闲方式时，CPU在执行中断服务子程序中查询GF0或GF1的状态，便可以判别出在发生中断时CPU是否处于空闲状态。

退出空闲方式的另一种方法是硬件复位，因为空闲方式时振荡器仍然在工作，所以只需要两个机器周期便可完成复位。RST引脚上的复位信号直接清“0”PCON.0位，从而使器件退出空闲工作方式，CPU从激活空闲方式指令的下一条指令开始执行程序。应用空闲方式时需要注意，激活空闲方式指令的下一条指令不能是对口的操作指令和对外部RAM的写入指令，以防止硬件复位过程中对外部RAM的误操作。

退出掉电方式的唯一方法是硬件复位。复位后，单片机内部特殊功能寄存器的内容被初始化，PCON=0，从而退出掉电方式。

5.3.3 低功耗方式应用编程

例5-4 利用外部中断将单片机从空闲模式下唤醒。Proteus仿真电路如图5.19所示。单片机的IO端口控制74HC595驱动8位数码管，定时器T0采用方式2即8位自动重装方式产生1毫秒

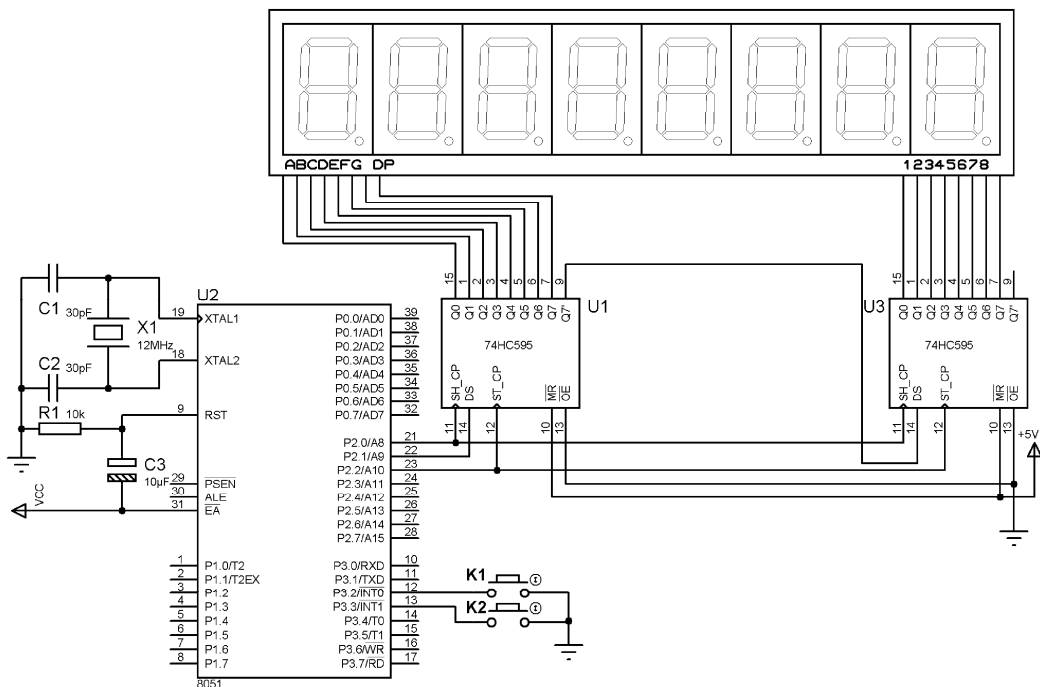


图5.19 利用外部中断将单片机从空闲模式下唤醒的Proteus仿真电路

定时,用于数码管扫描显示。上电后,右边5个数码管显示秒计数,计数范围为0~10000,显示10秒后,单片机进入空闲方式,数码管熄灭。按一下K1 ($\overline{\text{INT0}}$)或K2 ($\overline{\text{INT1}}$)按键可将单片机唤醒,继续显示秒计数,10秒后再次进入空闲模式。如果持续按下K1或K2按键(即 $\overline{\text{INT0}}$ 或 $\overline{\text{INT1}}$ 中任一个或两个同时为低电平),则单片机不会进入空闲模式,直到 $\overline{\text{INT0}}$ 和 $\overline{\text{INT1}}$ 都为高电平为止。

C51源程序文件如下。

```
#include "reg51.h"
#include "intrins.h"
#define MAIN_Fosc 12000000L //定义主时钟
#define uchar unsigned char
#define uint unsigned int
#define DIS_DOT 0x20
#define DIS_BLACK 0x10
#define DIS_ 0x11
#define LED_TYPE 0x00 //定义数码管类型, 0x00--共阴, 0xff--共阳
#define Timer0_Reload(65536UL - (MAIN_Fosc/1000))//T0中断频率, 1000次/秒

uchar code t_display[]={ //标准字库
// 0 1 2 3 4 5 6 7 8 9 A B C D E F
0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x39,
0x5E,0x79,0x71,
//black - H J K L No P U t G Q r M y
0x00,0x40,0x76,0x1E,0x70,0x38,0x37,0x5C,0x73,0x3E,0x78,0x3d,0x67,
0x50,0x37,0x6e,
0xBF,0x86,0xDB,0xCF,0xE6,0xED,0xFD,0x87,0xFF,0xEF,0x46}; //0.1.2.3.
4.5.6.7.8.9.-1
uchar code T_COM[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80}; //位码

/***** 595的IO端口定义 *****/
sbit DS = P2^1; // data input
sbit ST_CP = P2^2; // RCLK store (latch) clock
sbit SH_CP = P2^0; // Shift data clock

/***** 本地变量声明 *****/
uchar LED8[8]; //显示缓冲
uchar display_index; //显示位索引
bit B_1ms; //1ms标志
uint msecond;
uint Test_cnt; //测试用的秒计数变量
uchar SleepDelay; //唤醒后再进入睡眠所延时的时间
uchar B_100us; //100us标志

/***** 显示计数函数 *****/
void Display(void){
uchar i;
LED8[3] = Test_cnt / 10000;
LED8[4] = (Test_cnt % 10000) / 1000;
LED8[5] = (Test_cnt % 1000) / 100;
LED8[6] = (Test_cnt % 100) / 10;
```

```

    LED8[7] = Test_cnt % 10;
    for(i=3; i<7; i++){          //消隐无效0
        if(LED8[i] > 0) break;
        LED8[i] = DIS_BLACK;
    }
}

/***** 主函数 *****/
void main(void){
    uchar i;
    display_index = 0;
    for(i=0; i<8; i++) LED8[i] = DIS_BLACK; //全部消隐
    TMOD = 0x02;                          //T0工作8位自动重装方式
    TH0 = 0x9c;                            //设置100μs定时初值
    TL0 = 0x9c;
    EA = 1;                                //开中断
    ET0 = 1;
    TR0 = 1;                                //启动T0
    Test_cnt = 0;                           //秒计数范围为0~10000
    Display();                               //显示秒计数
    SleepDelay = 0;
    while(1) {
        if(B_1ms){                         //1ms到
            B_1ms = 0;
            if(++msecond >= 1000){          //1秒到
                msecond = 0;                //清1000ms计数
                Test_cnt++;                  //秒计数+1
                if(Test_cnt > 10000) Test_cnt = 0; //秒计数范围为0~10000
                Display();                  //显示秒计数
                if(++SleepDelay >= 10){     //10秒后睡眠
                    SleepDelay = 10;        //避免长按溢出
                    if(INT0 && INT1){        //INT0、INT1都是高电平才进入休眠
                        SleepDelay = 0;
                        TR0 = 0;            //停止T0
                        DS = 0;
                        for(i=0; i<16; i++){ //关闭显示, 省电
                            SH_CP = 1;
                            SH_CP = 0;
                        }
                        ST_CP = 1;
                        ST_CP = 0;          //锁存输出数据
                        IE1 = 0;            //INT1标志位
                        IE0 = 0;            //INT0标志位
                        EX1 = 1;            //INT1允许
                        EX0 = 1;            //INT0允许
                        IT0 = 1;            //INT0下降沿中断
                        IT1 = 1;            //INT1下降沿中断
                        EA = 1;            //开中断
                        PCON |= 0x01;      //进入空闲模式
                        _nop_();
                        _nop_();

```

```

        _nop_();
        TR0 = 1;           //重新启动T0
    }
}

}

}

}

}

/****** INT0中断函数 *****/
void INT0_int (void) interrupt 0{   //进中断时已经清除标志
    EX0 = 0;
    IE0 = 0;
}

/****** INT1中断函数 *****/
void INT1_int (void) interrupt 2{   //进中断时已经清除标志
    EX1 = 0;
    IE1 = 0;
}

/****** 向HC595发送一个字节函数 *****/
void Send_595(uchar dat){
    uchar i;
    for(i=0; i<8; i++){
        dat <<= 1;
        DS = CY;
        SH_CP = 1;
        SH_CP = 0;
    }
}

/****** 显示扫描函数 *****/
void DisplayScan(void){
    Send_595(~LED_TYPE ^ T_COM[display_index]);          //输出位码
    Send_595(LED_TYPE ^ t_display[LED8[display_index]]); //输出段码
    ST_CP = 1;
    ST_CP = 0;                                           //锁存输出数据
    if(++display_index >= 8)      display_index = 0;     //8位结束回0
}

/****** T0 1ms中断函数 *****/
void timer0 (void) interrupt 1{
    B_100μs++;
    If(B_100μs==10){
DisplayScan();       //1ms扫描显示一位
        B_1ms = 1;         //1ms标志
        B_100μs=0;
    }
}

```


键盘与显示器接口应用

6.1 LED显示器接口技术

发光二极管LED (Light Emitting Diode) 是单片机应用系统中简单而常用的输出设备，通常用来指示机器的状态或其他信息。它的优点是价格低、寿命长，对电压、电流的要求低及容易实现多路等，因而在单片机应用系统中获得广泛的应用。

LED是近似于恒压的组件，导电时（发光）的正向压降一般约为1.6V或2.4V，反向击穿电压一般 $\geq 5V$ 。工作电流通常为10~20mA，故电路中需串联适当的限流电阻。发光强度基本上与正向电流成正比。发光效率和颜色取决于制造的材料，一般常用红色，偶尔也用黄色或绿色。多个LED可接成共阳极或共阴极形式。图6.1为LED共阳极连接，通过驱动器接到系统的并行输出口上，由CPU输出适当的代码来点亮或熄灭相应的LED。

发光二极管显示器驱动（点亮）的方法有两种。一种是静态驱动法，即给欲点亮的LED通以恒定的电流。这种驱动方法要有寄存器、译码器、驱动电路等逻辑部件。当需要显示的位数增加时，所需的逻辑部件及连线也相应增加，成本也增加。另一种是动态驱动方法。这种方法是给欲点亮的LED通以脉冲电流，此时LED的亮度是通、断的平均亮度。为保证亮度，通过LED的脉冲电流应数倍于其额定电流值。利用动态驱动法可以减少需要的逻辑部件和连线。单片机应用系统常采用动态驱动方法。

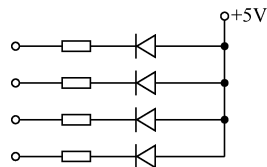


图 6.1 LED 的共阳极连接

6.1.1 七段LED数码管显示器

最常用的一种数码管显示器是由7段条形的LED组成的，如图6.2所示。点亮适当的字段，就可显示出不同的数字。此外，不少七段数码管显示器在右下角带有一个圆形的LED作为小数点，这样一共有8段，恰好适用于8位的并行系统。

图6.2 (a) 为共阴极接法，公共阴极接地，当各段阳极上的电平为“1”时，该段点亮，电平为“0”时，该段熄灭；图6.2 (b) 为共阳极接法，公共阳极接+5V电源，当各段阴极上的电

平为“0”时，该段就点亮，电平为“1”时，该段就熄灭。图中R是限流电阻。图6.2（c）为七段LED数码管显示器内部段的排列。

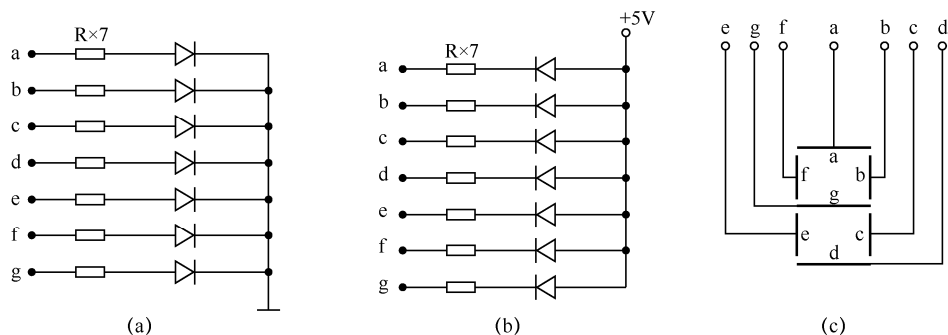


图6.2 七段LED数码管显示器的连接

为了在七段LED上显示不同的数字或字符，首先要把数字或字符转换成相应的段码（又称字形码），由于电路接法不同，因此形成的段码也不相同，见表6.1。

表6.1 七段数码管显示器的段码表

存储器地址	显示数字	D ₇ D ₆ D ₅ D ₄ D ₃ D ₂ D ₁ D ₀ g f e d c b a	共阴极接法 段码（十六进制数）	共阳极接法 段码（十六进制数）
SEG	0	0 0 1 1 1 1 1 1	3F	40
SEG+1	1	0 0 0 0 0 1 1 0	06	79
SEG+2	2	0 1 0 1 1 0 1 1	5B	24
SEG+3	3	0 1 0 0 1 1 1 1	4F	30
SEG+4	4	0 1 1 0 0 1 1 0	66	19
SEG+5	5	0 1 1 0 1 1 0 1	6D	12
SEG+6	6	0 1 1 1 1 1 0 1	7D	02
SEG+7	7	0 0 0 0 0 1 1 1	07	78
SEG+8	8	0 1 1 1 1 1 1 1	7F	00
SEG+9	9	0 1 1 0 0 1 1 1	67	18
SEG+10	A	0 1 1 1 0 1 1 1	77	08
SEG+11	B	0 1 1 1 1 1 0 0	7C	03
SEG+12	C	0 0 1 1 1 0 0 1	39	46
SEG+13	D	0 1 0 1 1 1 1 0	5E	21
SEG+14	E	0 1 1 1 1 0 0 1	79	06
SEG+15	F	0 1 1 1 0 0 0 1	71	0E

将显示数字或字符转换成段码的过程可以通过硬件译码或软件译码来实现。图6.3为采用硬件译码BCD数码管显示器与8051单片机的接口电路。这种显示器内部集成了硬件段译码器，能自动将输入的BCD数转换成七段LED段码，直接点亮显示器的段。例6-1是采用C51编写的驱动程序，执行程序后可以看到数码管循环显示数字“12345678”。

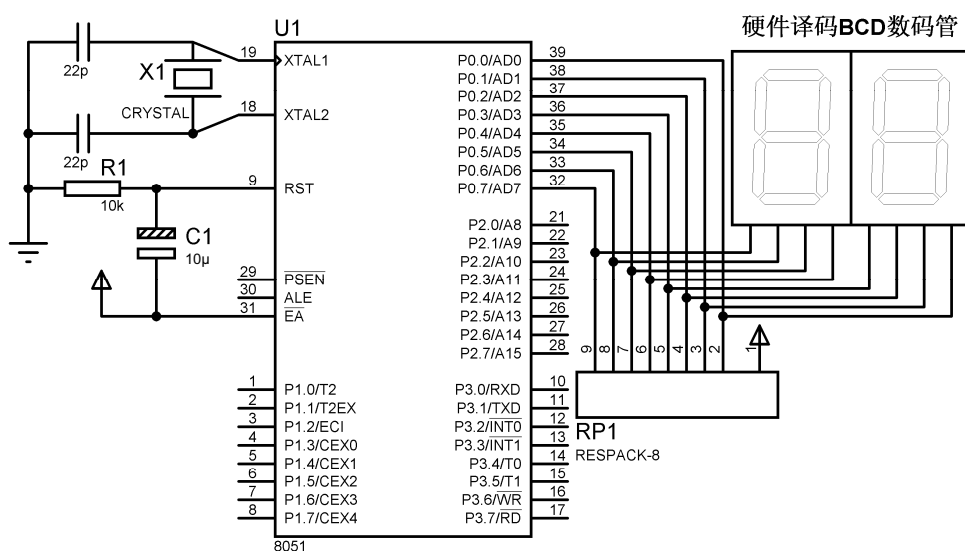


图6.3 采用硬件译码BCD数码管显示器与8051单片机的接口电路

例6-1 BCD数码管C51驱动程序。

```
#include<reg52.h>
/***** 延时函数 *****/
void delay() {
    unsigned int i;
    for(i=0;i<35000;i++);
}

/***** 主函数 *****/
main() {
    while(1) {
        P0=0x12; delay(); //从P0口输出BCD码12
        P0=0x34; delay(); //从P0口输出BCD码34
        P0=0x56; delay(); //从P0口输出BCD码56
        P0=0x78; delay(); //从P0口输出BCD码78
    }
}
```

带有硬件译码器的BCD数码管驱动简单，但价格较高，如果要显示多位数字或字符时，则采用如图6.3所示的接口无论从成本还是从耗电量来说都是不太合适。为此可以采用普通7段LED数码管，根据数码管的连接方式排出表6.1所列的显示段码表，在驱动程序中利用软件查表方式进行译码。图6.4为单个7段LED数码管与8051单片机的接口电路。

例6-2 单个数码管软件译码C51驱动程序。

```
#include<reg52.h>
#define uchar unsigned char
```

```

#define uint unsigned int

uchar code SEG[]={0x3f,0x06,0x5b,0x4f,0x66, //段码表
                  0x6d,0x7d,0x07,0x7f,0x6f};

/***** 延时函数 *****/
void delay(){
    uint i;
    for(i=0;i<35000;i++);
}

/***** 主函数 *****/
main(){
    while(1){
        uchar i;
        for(i=0;i<10;i++){
            P0=table[i];
            delay();
        }
    }
}

```

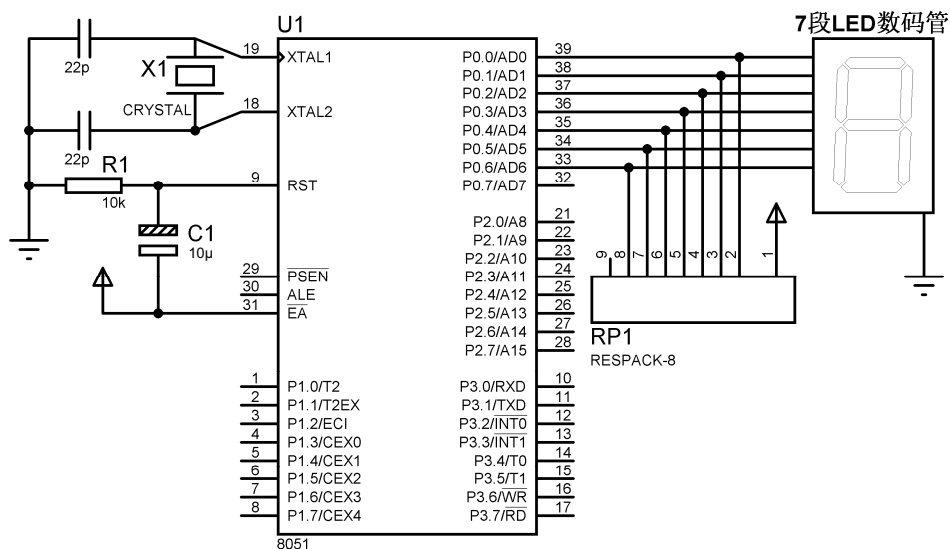


图6.4 单个7段LED数码管与8051单片机的接口电路

图6.5为多位7段LED数码管与8051单片机的接口电路，采用软件译码和动态扫描显示技术。设计思想是根据要显示的数字或字符去查表取得相应的段码，具体显示时，采用逐位扫描的方法控制哪一位数码管被点亮，在本接口中先从最左一位数码管开始，逐个左移，直至最后一个数码管显示完毕，然后重复上述过程。由于人眼的视觉暂留，因此看起来不会有闪动的感觉。

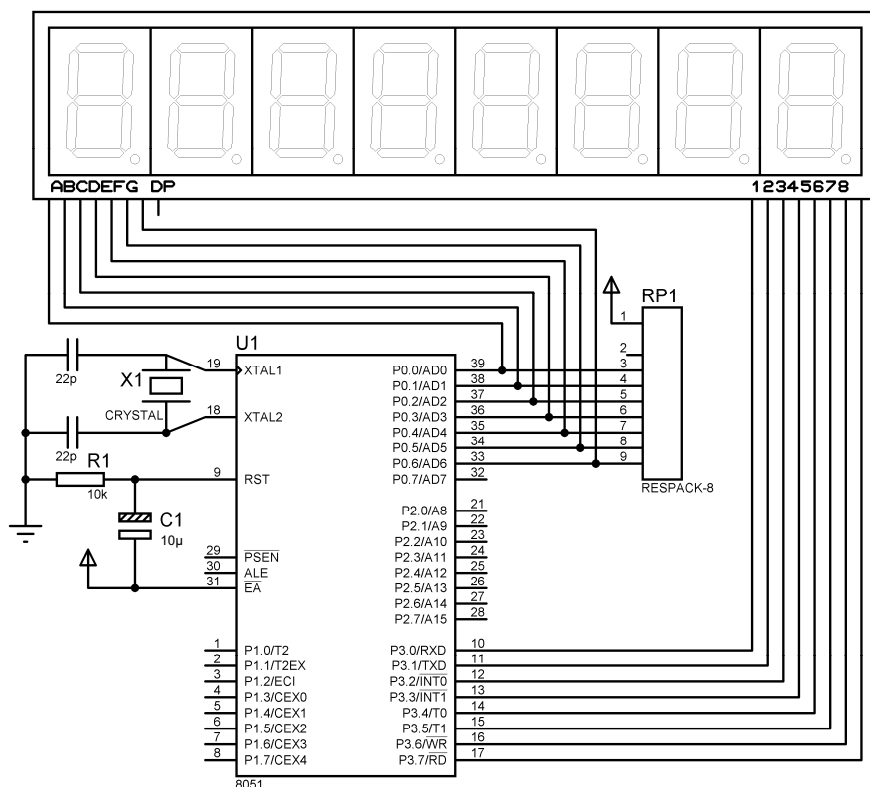


图6.5 多位7段LED数码管与8051单片机的接口电路

例6-3 多位数码管动态扫描C51驱动程序。

```
#include<reg52.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int
uchar code SEG[]={0x3f,0x06,0x5b,0x4f,0x66, //段码表
                  0x6d,0x7d,0x07,0x7f,0x6f};

/***** 延时函数 *****/
void delay(){
    uint i;
    for(i=0;i<1000;i++);
}

/***** 主函数 *****/
main(){
    while(1){
        uchar i;
        P3=0x7f;
        for(i=0;i<8;i++){
            P3=_crol_(P3,1);
```

```

        P0= SEG[i];
        delay();
    }
}
}

```

6.1.2 单个74HC595驱动多位LED数码管

74HC595是一款漏极开路输出的CMOS移位寄存器，具有标准SPI串行接口，可以串行级联使用。74HC595的引脚排列如图6.6所示，各引脚功能见表6.2。

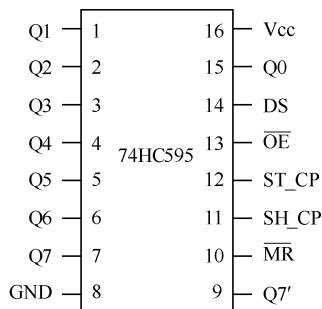


图6.6 74HC595的引脚排列

表6.2 74HC595的引脚功能

引 脚	功 能
DS	串行数据输入引脚
Q0~Q7	三态输出引脚
Q7'	串行数据输出引脚，用于级联
SH_CP	移位寄存器时钟输入端
ST_CP	存储寄存器时钟输入端
MR	复位端
OE	输出允许
Vcc	电源端
GND	接地端

74HC595与数据相关的引脚包括：

DS：串行数据输入，接单片机的某个I/O引脚。

Q0~Q7：8位并行数据输出，可以直接控制8个LED，或者是LED数码管的8个段端。

Q7'：级联输出端，与下一个74HC595的DS相连，实现多个芯片之间的级联。

74HC595与控制相关的引脚包括：

SH_CP：时钟输入。上升沿时移位寄存器中的数据依次移动一位，即Q0中的数据移到Q1中，Q1中的数据移到Q2中，依次类推。下降沿时移位寄存器中的数据保持不变。

ST_CP：存储寄存器的时钟输入。上升沿时移位寄存器中的数据进入存储寄存器，下降沿时存储寄存器中的数据保持不变。应用时，通常将ST_CP置为低电平，移位结束后，再在

ST_CP端产生一个正脉冲以更新显示数据。

$\overline{\text{MR}}$ ：复位端。低电平时将移位寄存器中的数据清“0”，通常将它直接连高电平（Vcc）。

$\overline{\text{OE}}$ ：输出允许，高电平时禁止输出（高阻态）。实际应用时可以将它直接连低电平（GND），也可以用单片机的一个引脚来控制它，以便产生闪烁和熄灭的效果。

74HC595是串入并出、带有锁存功能的移位寄存器，在移位过程中，输出端的数据可以保持不变，这在串行传输速度慢的场合很有用处。74HC595功能真值表见表6.3。

表6.3 74HC595的功能真值表

输 入					输 出
DS	SH_CP	$\overline{\text{MR}}$	ST_CP	$\overline{\text{OE}}$	
×	×	×	×	H	Q0~Q7输出高阻
×	×	×	×	L	Q0~Q7输出有效值
×	×	L	×	×	移位寄存器清0
L	↑	H	×	×	移位寄存器存储低电平
H	↑	H	×	×	移位寄存器存储高电平
×	↓	H	×	×	移位寄存器状态保持
×	×	×	↑	×	输出存储器锁存移位寄存器中的状态值
×	×	×	↓	×	输出存储器状态保持

74HC595的使用方法很简单。使用时， $\overline{\text{MR}}$ 接高电平， $\overline{\text{OE}}$ 接低电平。从DS端输入数据，每输入一位，移位寄存器输入时钟SH_CP上升沿有效一次，直到8位数据输入完毕；然后，存储寄存器输入时钟ST_CP上升沿有效一次，输入的数据就被送到了输出端。通过SH_CP时钟上升沿将数据移入和通过ST_CP时钟上升沿将数据输出，是两个独立的过程，实际应用时互不干扰，在输出数据的同时可以移入数据。

图6.7为采用74HC595驱动6个LED数码管的原理图，用8051单片机的P3.0、P3.1和P3.2分别控制74HC595的SH_CP、DS和ST_CP，将 $\overline{\text{MR}}$ 和 $\overline{\text{OE}}$ 分别接Vcc和地。执行例6-4程序后，可以看到数码管上显示数字“123456”。

例6-4 74HC595驱动多位数码管的C51程序。

```
#include <reg51.h>
#include <intrins.h>
#define uchar unsigned char
#define NOP _nop_()

sbit SH_CP = P3^0;
sbit DS = P3^1;
sbit ST_CP = P3^2;

uchar ledcode[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
uchar ledbitselect[] = {0x0fe,0xfd,0xfb,0xf7,0xef,0xdf,0xbf,0x7f};
uchar i;
```

```

/***** 串行口初始化函数 *****/
void InitSerialPort(void){
    DS = 0;
    SH_CP = 0;
    ST_CP = 0;
}

/***** 延时函数 *****/
void delay(void){
    uchar jj;
    for(jj=0;jj<200;jj++){
        while(jj--);
    }

/***** 串行数据输入函数 *****/
void SerialSendData(uchar dat){
    uchar ii;
    uchar DSt=dat;
    for(ii=0;ii<8;ii++){
        if(DSt&0x80)DS=1;
        else DS = 0;
        DSt<<=1;
        SH_CP =0;
        NOP; NOP;NOP; NOP;
        SH_CP = 1;
        NOP; NOP;
    }
    ST_CP = 1;
    NOP; NOP;NOP; NOP;
    ST_CP = 0;
}

/***** 主函数 *****/
void main(){
    InitSerialPort();
    while(1){
        delay();
        delay();
        P2 = ledbitselect[i];
        SerialSendData(ledcode[i]);
        i=(i+1)%8;
    }
}

```

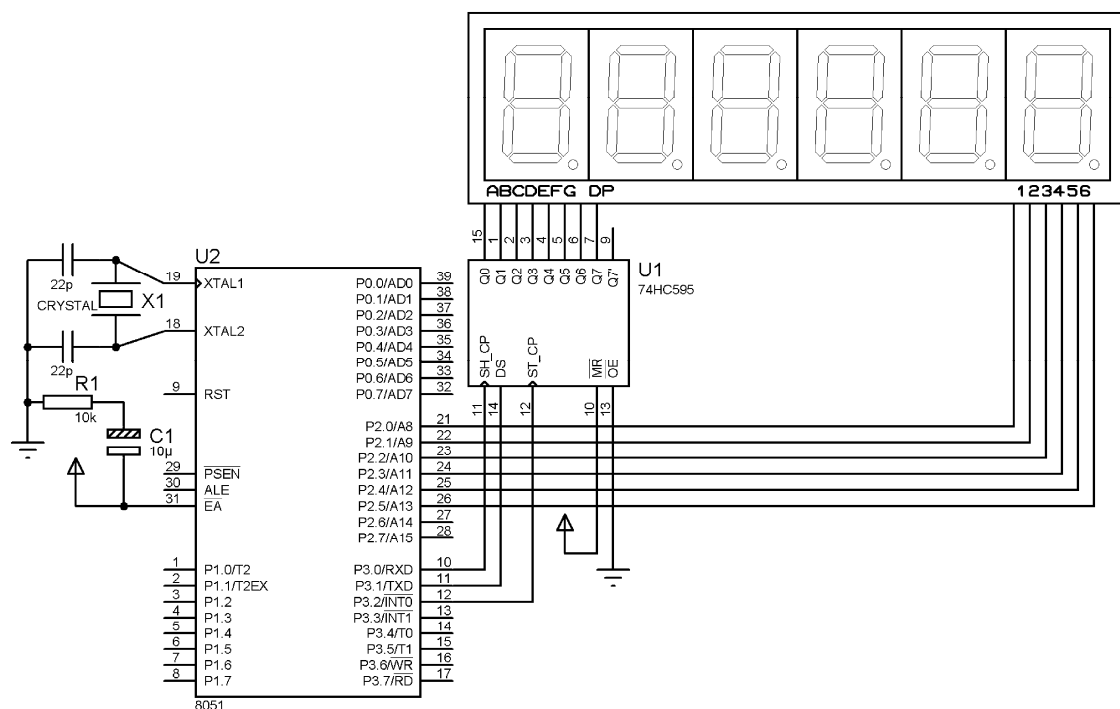



图6.7 采用74HC595驱动6个LED数码管的原理图

6.1.3 串行接口8位共阴极LED驱动器MAX7219

MAX7219是MAXIM公司生产的一种串行接口7段共阴极LED数码管显示驱动器。其片内包含有一个BCD码到B码的译码器、多路复用扫描电路、字段和字位驱动器及存储每个数字的8×8 RAM，每位数字都可以被寻址和更新，允许对每一位数字选择B码译码或不译码。采用三线串行方式与MCU接口，电路十分简单，只需要一个10kΩ左右的外接电阻来设置所有LED数码管的段电流。

MAX7219的引脚排列如图6.8所示，各引脚功能如见表6.4。

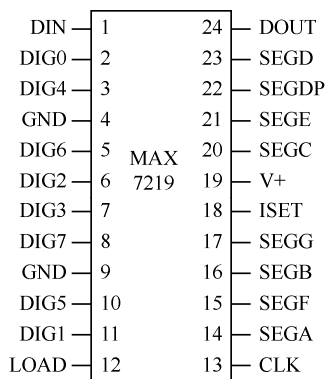


图6.8 MAX7219 的引脚排列

表6.4 MAX7219各引脚功能

引 脚	功 能
DIN	串行数据输入。在CLK时钟的上升沿，串行数据被移入内部移位寄存器，移入时最高位（MSB）在前
DIG0~DIG7	8根字位驱动引脚，从LED显示器吸入电流
GND	地，两根GND引脚必须相连
CLK	时钟输入，是串行数据输入时所需的移位脉冲。最高时钟频率为10MHz，在CLK的上升沿串行数据被移入内部移位寄存器，在CLK的下降沿数据从DOUT移出
SEGa~g, dp	7段和小数点驱动输出，提供LED显示器源电流
ISSET	通过一个10kΩ电阻R _{SET} 接到V+以设置峰值段电流
V+	+5V电源电压
DOUT	串行数据输出。输入到DIN的数据经过16.5个时钟周期后，在DOUT端有效

MAX7219采用串行数据传输方式，由16位数据包发送到DIN引脚的串行数据在每个CLK的上升沿被移入到内部16位移位寄存器中，然后在LOAD的上升沿将数据锁存到数字或控制寄存器中。LOAD信号必须在第16个时钟上升沿同时或之后，但在下一个时钟上升沿之前变高，否则将会丢失数据。DIN端的数据通过移位寄存器传送，并在16.5个时钟周期后出现在DOUT端。DOUT端的数据在CLK的下降沿输出。串行数据以16位为一帧。其中，D15~D12可以任意，D11~D8为内部寄存器地址，D7~D0为寄存器数据，格式见表6.5。

表6.5 MAX7219的串行数据格式

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
×	×	×	×	地 址				MSB 数 据 LSB							

MAX7219的数据传输时序如图6.9所示。

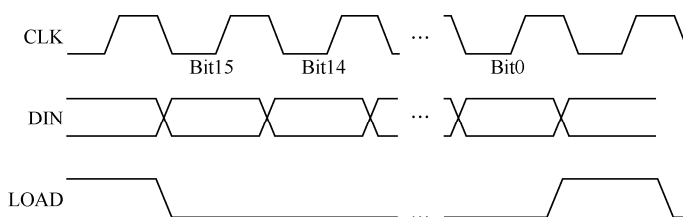


图6.9 MAX7219的数据传输时序

MAX7219具有14个可寻址的内部数字和控制寄存器，8个数字寄存器由一个片内8×8双端口SRAM实现。它们可以直接寻址，因此可以对单个数字进行更新，并且只要V₊超过2V，数据就可以保留下去。控制寄存器有5个，分别为译码方式、显示亮度、扫描界限（扫描数位的个数）、停机和显示测试。另外还有一个空操作寄存器（NO-OP），在不改变显示或影响任一控制寄存器的条件下，在器件级联时，允许数据从DIN传送到DOUT。表6.6为MAX7219的内部寄存器及其地址。

表6.6 MAX7219的内部寄存器及其地址

寄 存 器	地 址					
	D15~D12	D11	D10	D9	D8	十六进制代码
NO-OP	×	0	0	0	0	×0H
数字0	×	0	0	0	1	×1H
数字1	×	0	0	1	0	×2H
数字2	×	0	0	1	1	×3H
数字3	×	0	1	0	0	×4H
数字4	×	0	1	0	1	×5H
数字5	×	0	1	1	0	×6H
数字6	×	0	1	1	1	×7H
数字7	×	1	0	0	0	×8H
译码方式	×	1	0	0	1	×9H
亮度	×	1	0	1	0	×AH
扫描界限	×	1	0	1	1	×BH
停机	×	1	1	0	0	×CH
显示测试	×	1	1	1	1	×FH

下面用表格形式对MAX7219内部寄存器中不同数据所表示的含义进行说明。表6.7为译码方式寄存器中数据的含义。从表中可见，寄存器中的每一位与一个数字位相对应，逻辑高电平选择B码译码，而逻辑低电平则旁路译码器。

表6.7 译码方式寄存器（地址 = ×9H）中数据的含义

含 义	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
7~0位均不译码	0	0	0	0	0	0	0	0	00H
0位译成B码，7~1均不译码	0	0	0	0	0	0	0	1	01H
3~0位译成B码，7~4均不译码	0	0	0	0	1	1	1	1	0FH
7~0位均译成B码	1	1	1	1	1	1	1	1	FFH

MAX7219可用引脚V+和引脚ISET之间所接外部电阻 R_{SET} 来控制显示亮度。来自段驱动器的峰值电流通常为进入ISET电流的100倍。 R_{SET} 既可为固定电阻，也可为可变电阻，以提供来自面板的亮度调节。其最小值为9.52k Ω 。段电流的数字控制由内部脉宽调制DAC控制。该DAC通过亮度寄存器向低4位加载。该DAC将平均峰值电流按16级比例设计，从 R_{SET} 设置峰值电流31/32的最大值到1/32的最小值，见表6.8。最大亮度出现在占空比为31/32时。

表6.8 亮度寄存器（地址 = ×AH）

占空比（亮度）	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
1/32（最小亮度）	×	×	×	×	0	0	0	0	×0H
3/32	×	×	×	×	0	0	0	1	×1H
5/32	×	×	×	×	0	0	1	0	×2H
...
29/32	×	×	×	×	1	1	1	0	×EH
31/32（最大亮度）	×	×	×	×	1	1	1	1	×FH

扫描界限寄存器用于设置所显示的数字位, 可以从1~8, 通常以扫描速率为1300Hz、8位数字、多路方式显示。因为所扫描数字的多少会影响显示亮度, 所以要注意调整。如果扫描界限寄存器被设置为3个数字或更少, 则各数字驱动器将消耗过量的功率。因此 R_{SET} 电阻的值必须按所显示数字的位数多少适当调整, 以限制各个数字驱动器的功耗。表6.9为扫描界限寄存器中数据的含义。

表6.9 扫描界限寄存器(地址 = $\times BH$) 中数字的含义

显示数字位	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
只显示第0位数字	×	×	×	×	×	0	0	0	$\times 0H$
显示第0位~第1位数字	×	×	×	×	×	0	0	1	$\times 1H$
显示第0位~第2位数字	×	×	×	×	×	0	1	0	$\times 2H$
...
显示第0位~第6位数字	×	×	×	×	×	0	1	1	$\times 6H$
显示第0位~第7位数字	×	×	×	×	×	1	1	1	$\times 7H$

当MAX7219处于停机方式时, 扫描振荡器停止工作, 所有的段电流源被拉到地, 而所有的位驱动器被拉到 $V+$, 此时LED将不显示。在数字和控制寄存器中的数据保持不变。停机方式可用于节省功耗或使LED处于闪烁。MAX7219退出停机方式的时间不到250微秒, 在停机方式下, 显示驱动器还可以编程, 停机方式可以被显示测试功能取消。表6.10为停机寄存器中数据的含义。

表6.10 停机寄存器(地址 = $\times CH$) 中数据的含义

工作方式	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
停机	×	×	×	×	×	×	×	0	$\times 0H$
正常	×	×	×	×	×	×	×	1	$\times 1H$

显示测试寄存器有两种工作方式: 正常和显示测试。在显示测试方式下, 8位数字被扫描, 占空比为31/32, 通过不考虑(但不改变)所有控制寄存器和数据寄存器(包括停机寄存器)内的控制字来接通所有的LED显示器。表6.11为显示测试寄存器中数据的含义。

表6.11 显示测试寄存器(地址 = $\times FH$) 中数据的含义

工作方式	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
正常	×	×	×	×	×	×	×	0	$\times 0H$
显示测试	×	×	×	×	×	×	×	1	$\times 1H$

数字0~7寄存器受译码方式寄存器的控制: 译码或不译码。数字寄存器可将BCD码译成B码(0~9、-、E、H、L、P), 见表6.12。如果不译码, 则数字寄存器中数据的D6~D0位分别对应7段LED显示器的A~G段, D7位对应LED的小数点DP, 某一位数据为1, 则点亮与该位对应的LED段, 数据为0, 则熄灭该段。

表6.12 数字0~7寄存器 (地址 = ×1H~×8H)

7段字形	寄存器数据								点 亮 段							
	D7	D6	D5	D4	D3	D2	D1	D0	DP	A	B	C	D	E	F	G
0	×	×	×	×	0	0	0	0	×	1	1	1	1	1	1	0
1	×	×	×	×	0	0	0	1	×	0	1	1	0	0	0	0
2	×	×	×	×	0	0	1	0	×	1	1	0	1	1	0	1
3	×	×	×	×	0	0	1	1	×	1	1	1	1	0	0	1
4	×	×	×	×	0	1	0	0	×	0	1	1	0	0	1	1
5	×	×	×	×	0	1	0	1	×	1	0	1	1	0	1	1
6	×	×	×	×	0	1	1	0	×	1	0	1	1	1	1	1
7	×	×	×	×	0	1	1	1	×	1	1	1	0	0	0	0
8	×	×	×	×	1	0	0	0	×	1	1	1	1	1	1	1
9	×	×	×	×	1	0	0	1	×	1	1	1	1	0	1	1
-	×	×	×	×	1	0	1	0	×	0	0	0	0	0	0	1
E	×	×	×	×	1	0	1	1	×	1	0	0	1	1	1	1
H	×	×	×	×	1	1	0	0	×	0	1	1	0	1	1	1
L	×	×	×	×	1	1	0	1	×	0	0	0	1	1	1	0
P	×	×	×	×	1	1	1	0	×	1	1	0	0	1	1	1
暗	×	×	×	×	1	1	1	1	×	0	0	0	0	0	0	0

注：小数点DP由D7位控制，D7=1点亮小数点。

MAX7219可以级联使用，这时需要用到空操作寄存器（NO-OP）。空操作寄存器的地址为×0H。将所有级联器件的LOAD端连在一起，将DOUT端连接到相邻MAX7219的DIN端。例如，将4个MAX7219级联使用，在对第4片MAX7219写入时，发送所需要的16位字，其后跟3个空操作代码（×0××），当LOAD变高时，数据被锁存在所有器件中。前3个芯片接受空操作指令，第4个芯片将接受预期的数据。

图6.10为8031单片机与MAX7219的一种接口。8051的P3.5连到MAX7219的DIN端，P3.6连到LOAD端，P3.7连到CLK端，采用软件模拟方式产生MAX7219所需的工作时序。例6-5为C51驱动程序，程序执行后，在LED上显示8051字样。

例6-5 MAX7219的C51显示驱动程序。

```
#include <reg51.h>
#define uchar unsigned char
#define uint unsigned int

sbit DIN = 0xB5;
sbit LOAD = 0xB6;
sbit CLK = 0xB7;
uchar code LED_code_09[10]=// 定义显示数字0-9数组
    {0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,0x7F,0x7B};
uint code LED_code_L07[8]= // 定义显示位置L0-L3数组
    {0x0100,0x0200,0x0300,0x0400,0x0500,0x0600,0x0700,0x0800};
/*****向MAX7219发送命令函数 *****/
```

```

void sent_LED( uint n ){
    uint i;
    i = (uchar)( n );
    CLK = 0; LOAD = 0; DIN = 0;
    for ( i=0x8000; i>=0x0001; i=i>>1 ){
        if ( ( n & i ) == 0 ) DIN = 0; else DIN = 1;
        CLK = 1; CLK = 0;
    }
    LOAD = 1;
}

/***** MAX7219初始化函数 *****/
void MAX7219_init(){
    sent_LED( 0x0C01 ); // 置LED为正常状态
    sent_LED( 0x0A04 ); // 置LED亮度为9/32
    sent_LED( 0x0B07 ); // 置LED扫描范围DIGIT0-7
    sent_LED( 0x0900 ); // 置LED显示为不译码方式
}

/***** 清除MAX7219函数 *****/
void cls(){
    uint i;
    for (i=0x0100; i<=0x0800; i+=0x0100 ) sent_LED( i );
}

/***** 数字显示函数 *****/
void disp_09( uchar H, uchar n ){
    if(( n & 0x80 ) == 0 ){
        sent_LED( LED_code_L07[ H ] | LED_code_09[ n ] );
    }
    else{
        sent_LED( LED_code_L07[ H ] | LED_code_09[ n & 0x7F ] | 0x80 );
    }
}

/***** 主函数 *****/
void main(){
    MAX7219_init();
    cls();
    disp_09( 0x07,0xff);disp_09( 0x06,0xff);
    disp_09( 0x05,0x01);disp_09( 0x04,0x05);
    disp_09( 0x03,0x00);disp_09( 0x02,0x08);
    disp_09( 0x01,0xff);disp_09( 0x00,0xff);
    while(1);
}

```

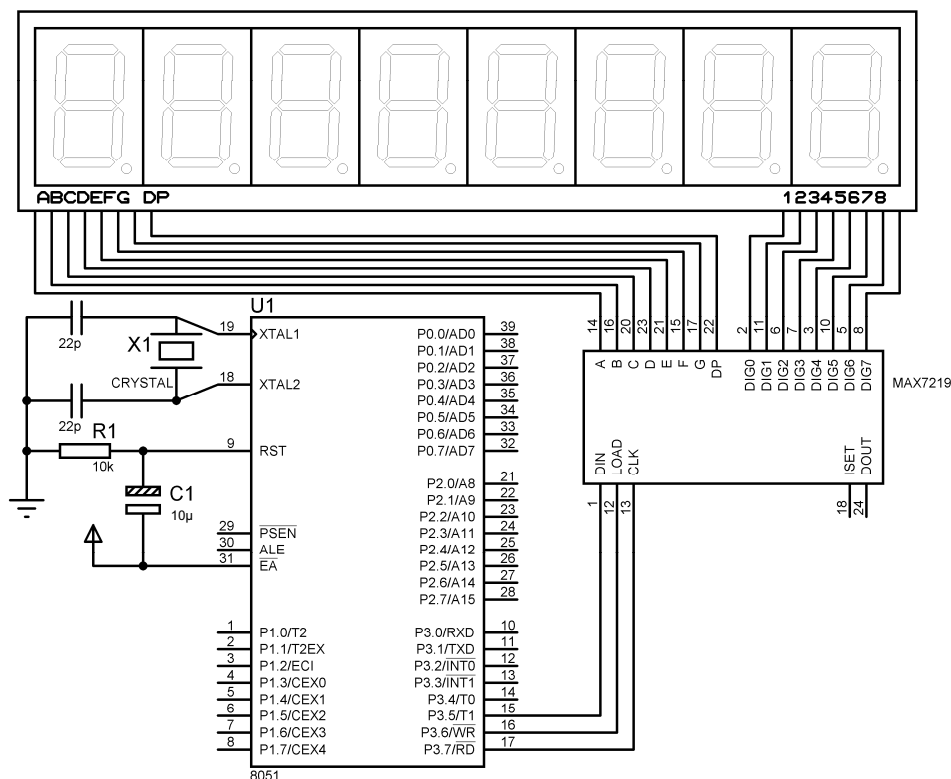


图6.10 8031与MAX7219单片机的一种接口

6.2 键盘接口技术

键盘是由一组按压式或触模式开关构成的阵列。键盘的设置由应用系统的具体功能来决定。键盘可分为编码式键盘和非编码式键盘。编码式键盘能够由硬件自动提供与被按键对应的其他编码，需要采用较多的硬件，价格较贵。非编码式键盘仅提供由行和列组成的矩阵，其硬件逻辑与按键编码不存在严格的对应关系，而要由软件程序来确定。非编码键盘的硬件接口简单，但是要占用较多的CPU时间。

任何键盘接口均要解决下述三个主要问题。

1. 按键识别

决定是否有键被按下，如有，则识别键盘中与被按键对应的编码。

2. 反弹跳

当按键开关的触点闭合或断开到其稳定，会产生一个短暂的抖动和弹跳，如图6.11（a）所示。这是机械式开关的一个共同性问题。消除由于键抖动和弹跳产生的干扰可采用硬件方法，也可采用软件延迟的方法。通常在键数较少时采用硬件方法，如可采用如图6.11（b）所示的R-S触发器。当键数较多时，则经常用软件延时的方法来反弹跳。对于按键前沿弹跳，

可在检出有键按下后，先执行一个延时20ms的子程序，待弹跳消失后，再次判断此按键是否仍然按下，如果是，则执行此按键的功能子程序；如果不是，则说明此次为按键弹跳，不执行按键功能子程序。对于后沿弹跳的处理与此类似。

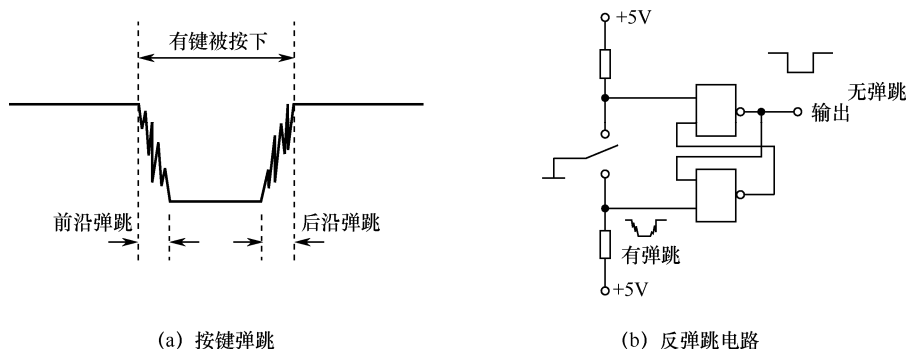


图6.11 按键弹跳和反弹跳电路

3. 串键保护

由于操作不慎，因此可能会造成同时有几个键被按下，这种情况被称为串键。有三种处理串键的技术：两键同时被按下、 n 键同时被按下和 n 键锁定。

“两键同时被按下”技术是在两个键同时被按下时产生保护作用。最简单的办法是当只有一个键被按下时才读取键值，最后仍被按下的是有效正确按键。另一种方法是当第一个按键未松开时，按第二个键不产生选通信号。

“ n 键同时被按下”技术或者不理睬所有被按下的键，直至只剩下一个键被按下时为止，或者将所有按键的信息都存入内部缓冲器中，然后逐个处理。

“ n 键锁定”技术只处理一个键，任何其他按下的键不产生任何键值。

6.2.1 编码键盘接口

键盘接口的这些任务可用硬件或软件来完成，相应地出现了两大类键盘，即编码式键盘和非编码式键盘。编码式键盘的基本任务是识别按键，提供按键读数。一个高质量的编码式键盘应具有反弹跳，处理同时按键等功能。目前已有用LSI技术制成的专用编码式键盘接口芯片。当按下某一按键时，该芯片能自动给出相应的编码信息，并可消除弹跳的影响，这样可使仪表设计者免除一部分软件编程，并可使CPU减轻用软件去扫描键盘的负担，提高CPU的利用率。

最简单的编码式键盘接口采用普通的编码器。图6.12 (a) 为采用8-3编码器 (74148) 作为键盘式编码器的静态编码键盘接口电路。每按一个键，在 A_2 、 A_1 、 A_0 端输出相应的按键读数，真值表列于图6.12 (b)。这种编码式键盘不进行扫描，因而被称为静态式编码器，缺点是一个按键需用一条引线，当按键增多时，引线将很复杂。

图6.13为利用8051单片机I/O端口实现的独立式键盘接口。这是一种最简单的编码式键盘结构，当有键被按下时，从单片机相应的端口引脚可以输入固定的电平值，采用查询方式工

作，要判断是否有键按下，用位处理指令十分方便。例6-6为这种键盘的C51驱动程序，执行后，可以看见接到P0口上的LED指示灯会随着按键被压下而闪动。

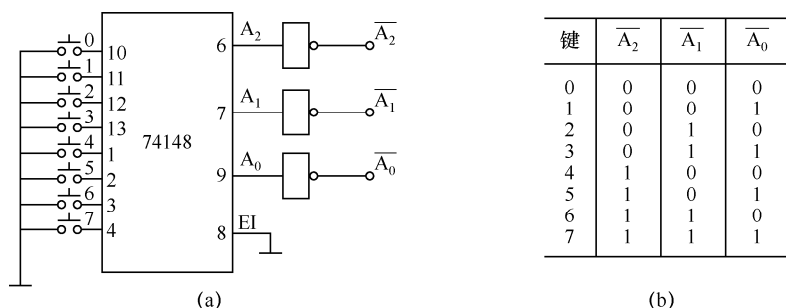


图6.12 静态式编码式键盘接口

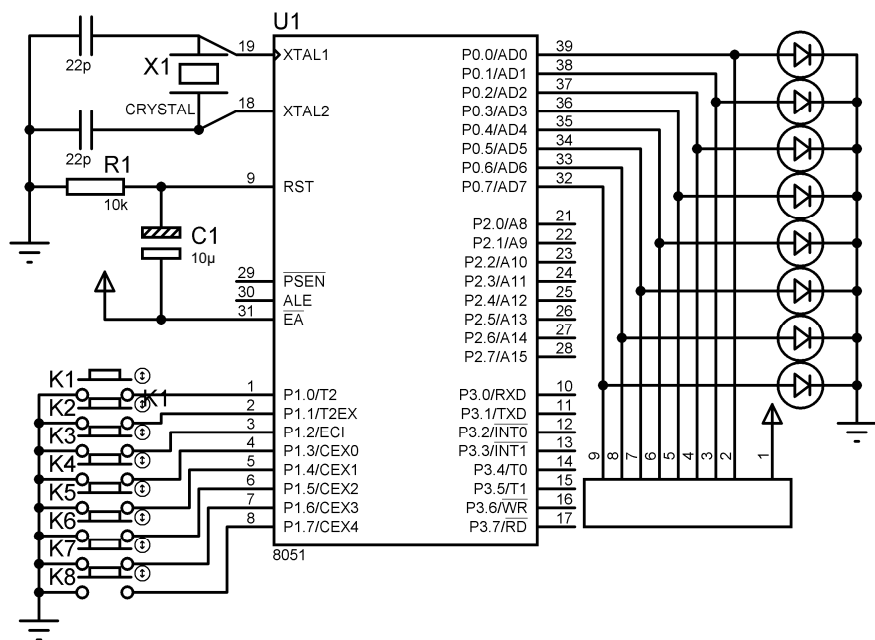


图6.13 利用8051单片机IO端口实现的独立式键盘接口

例6-6 用IO端口实现的独立键盘C51驱动程序。

```
#include<reg52.h>
#define uchar unsigned char
#define uint unsigned int
void delay(){          //延时
    unsigned int i;
    for(i=0;i<5000;i++);
}

void main(){
    uint temp;
    P1=0xff;
```

```

while(1){
    temp=P1 & 0xff;
    P0=temp;
    delay();
    P1=0xff;
}
}

```

6.2.2 非编码键盘接口

非编码式键盘大都采用按行、列排列的矩阵开关结构。这种结构可以减少硬件和连线。图6.14为4×4非编码式矩阵键盘的基本结构。

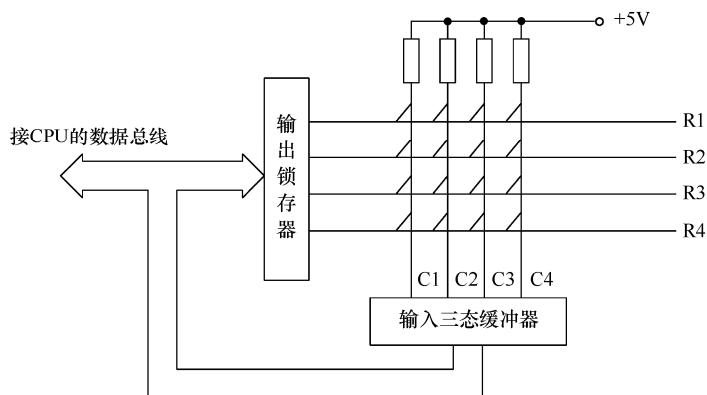


图6.14 4×4非编码式矩阵键盘的基本结构

在如图6.14所示接口电路中，输出锁存器的四根输出线分别与键盘的行线相连，列线电平信号经输入缓冲器送入单片机以进行按键识别。当输出锁存器的某一位为低电平时，位于该行的按键中若有一键被按下，则按下键的相应列线由于与行线短路而为低电平，否则为高电平。这样单片机就可以通过检查行线的输出电平和列线的输入电平来识别按键。矩阵键盘接口的设计思想是把键盘既作为输入又作为输出设备对待。行扫描法是一种常用的按键识别方法，采用步进扫描方式，CPU通过输出端口把一个步进的“0”逐行加至键盘的行线上，然后通过输入端口检查列线的状态，由行线和列线电平状态的组合来确定是否有键被按下，并确定被按键所处的行、列位置。图6.15为4×4矩阵键盘的行扫描按键识别原理图。

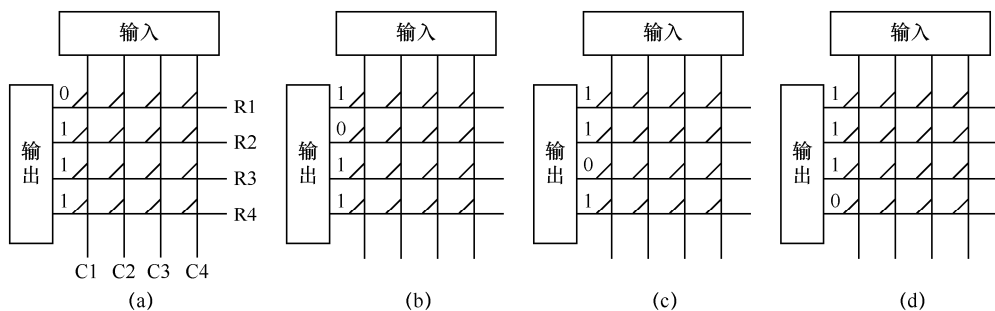


图6.15 4×4矩阵键盘的行扫描按键识别原理图

表6.13列出了识别图6.15中按键位置与各行、列之间的关系。其中，R1、R2、R3、R4表示行，C1、C2、C3、C4表示列。扫描第一行时，R1=0，若读入的列值C1=0，则表明按键K13被压下；如果C3=0，则表明按键K15被压下。第一行扫描完毕后，再扫描第二行，逐行扫描至最后一行为止，即可识别出所有的按键。

表6.13 识别图6.15中按键位置与各行、列之间的关系

R1	K13	K14	K15	K16
R2	K9	K10	K11	K12
R3	K5	K6	K7	K8
R4	K1	K2	K3	K4
	C1	C2	C3	C4

在实际应用中经常采用单片机的I/O端口实现矩阵键盘及LED数码管显示接口功能，具体电路如图6.16所示。单片机P3口用于矩阵键盘接口，P3.0~P3.3作为行扫描输出线，P3.4~P3.7作为读列输入线；P0口和P2口用于LED数码管显示接口，从P0口输出显示段码，P2口输出显示数位。例6-7是对应于图6.16接口电路的C51驱动程序，程序执行后，数码管上显示“01234567”，有键被按下时，数码管上将显示相应的字符。

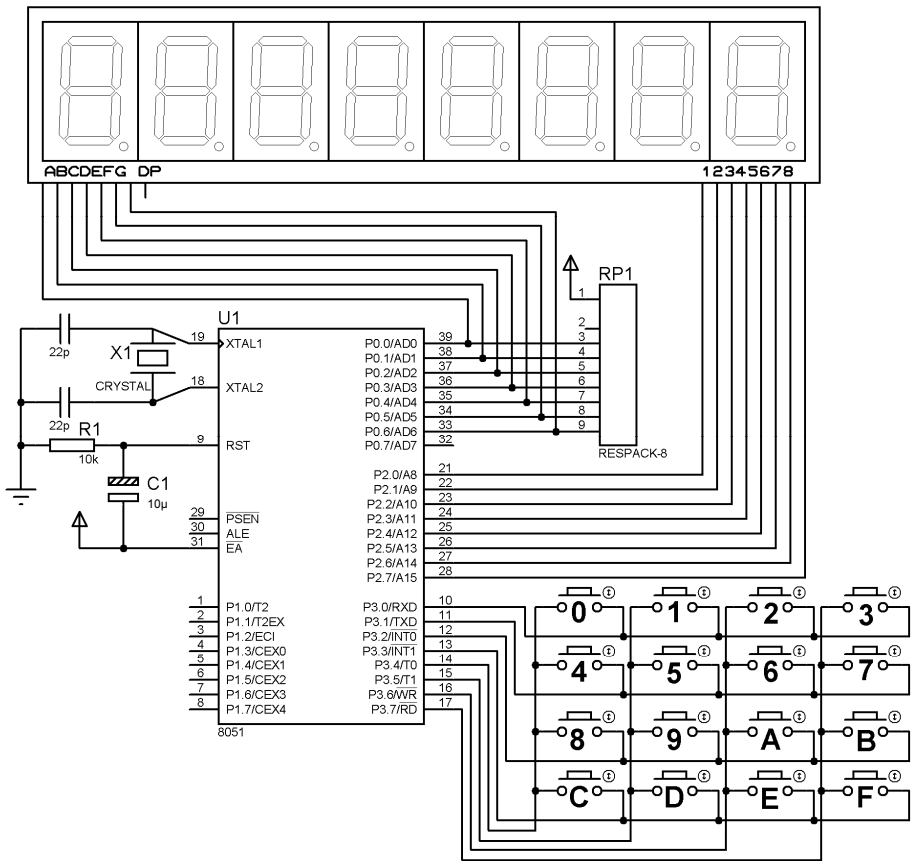


图6.16 采用单片机I/O端口实现的键盘显示接口电路

例6-7 对应于图6.16接口电路的C51驱动程序。

```

#include <reg51.h>
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int
uchar dspBf[8]={0,1,2,3,4,5,6,7}; //显示缓冲区
uchar code SEG[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07, //段码表
                  0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71,0x00};

/***** 数码管显示函数 *****/
void disp(){
    uchar i,dmask=0xfe;
    for(i=0;i<8;i++){
        P2=0xff; //熄灭所有LED
        P0=SEG[dspBf[i]];
        P2=dmask;
        dmask=_crol_(dmask,1); //修改扫描模式
    }
}

/***** 键盘扫描函数 *****/
uchar key(){
    uchar i,kscan;
    uchar temp=0x00,kval=0x00,kmask=0xfe;
    for(i=0;i<4;i++){
        P3=kmask; //扫描模式→P3口
        kscan=P3; //读P3口
        kscan=kscan>>4;
        switch(kscan&0x0f){
            case(0x0e):kval=0x00+temp; break;
            case(0x0d):kval=0x01+temp; break;
            case(0x0b):kval=0x02+temp; break;
            case(0x07):kval=0x03+temp; break;
            default:
                kmask=_crol_(kmask,1); //修改扫描模式
                temp=temp+0x04; break;
        }
    }
    if(kmask==0xef) kval=0x088;
    return kval;
}

/***** 主函数 *****/
void main(){
    uchar i,k;
    while(1){
        disp();
    }
}

```

```
k=key();
if(k!=0x88){
    dspBf[0]=k;
    for(i=1;i<8;i++){
        dspBf[i]=0x10;
    }
}
disp();
}
```

6.3 8279可编程键盘/显示器芯片接口技术

为了少占用CPU的工作时间，目前已经出现了专供键盘及显示器接口用的可编程接口芯片。Intel公司生产的8279可编程键盘/显示器接口芯片就是较为常见的一种。

6.3.1 8279的引脚排列

图6.17为8279芯片的管脚排列，各引脚功能见表6.14。

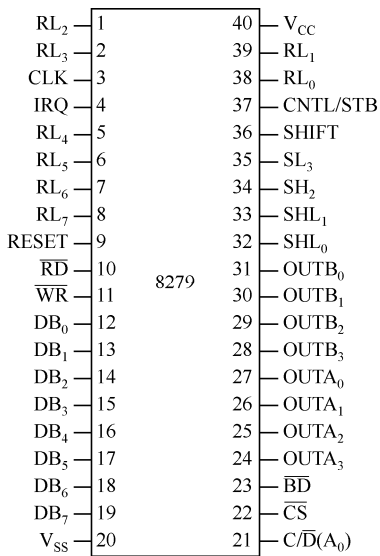


图6.17 8279芯片的引脚排列

表6.14 8279各引脚功能

引 脚	功 能
Vcc	+5V电源信号
Vss	地
DB ₀ ~DB ₇	双向数据总线
CLK	时钟输入，用于产生内部定时

(续表)

引脚	功能
RESET	复位信号
\overline{CS}	片选信号
C/\overline{D} (A0)	命令/数据区别信号, 高电平为命令, 低电平为数据
\overline{RD}	读信号
\overline{WR}	写信号
IRQ	中断请求输出
$SL_0 \sim SL_3$	扫描输出
$RL_0 \sim RL_7$	回馈输入
SHIFT	换挡输入
CNTL/STB	控制/选通输入
$OUTA_0 \sim OUTA_3$	显示输出A口信号
$OUTB_0 \sim OUTB_3$	显示输出B口信号
\overline{BD}	显示消隐输出

8279分为两个部分: 键盘部分和显示部分。键盘部分能够提供64按键阵列(可扩展为128)的扫描接口, 也可以接传感器阵列。键的按下可以是双键锁定或N键互锁。键盘输入经过反弹跳电路自动消除前后沿按键抖动影响之后, 被选通送入一个8字符的FIFO(先进先出栈)存储器。如果送入的字符多于8个, 则溢出状态置位。按键输入后将中断输出线升到高电平向CPU发中断申请。显示部分对7段LED、白炽灯或其他器件提供显示接口。8279有一个内部的16×8显示RAM, 组成一对16×4存储器。显示RAM可由CPU写入或读出。显示方式有从右进入的计算器方式和从左进入的电传打字方式。显示RAM每次读写之后, 其地址自动加1。

6.3.2 8279的数据输入、显示输出及命令格式

(1) 数据输入

数据输入有三种方式, 即键扫描方式、传感器扫描方式和选通输入方式。

采用键扫描方式时, 扫描线为 $SL_0 \sim SL_3$, 回馈线为 $RL_0 \sim RL_7$ 。每按下一个键, 便由8279自动编码, 并送入先进先出栈FIFO, 同时产生中断请求信号IRQ。键的编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
CNTL	SHIFT	扫描行序号			回馈线(列)序号		

如果芯片的控制脚CNTL和换挡脚SHIFT接地, 则D7和D6均为“0”, 如被按下键的位置在第2行(扫描行序号为010), 且与第4列回馈线(列序号为100)相交, 则该键所对应的代码为00010100, 即14H。

8279的扫描输出有两种方式: 译码扫描和编码扫描。所谓译码扫描, 即4条扫描线在同一时刻只有一条是低电平, 并且以一定的频率轮流更换。如果用户键盘的扫描线多于4, 则可采用编码输出方式。此时, $SL_0 \sim SL_3$ 输出的是从0000~1111的二进制计数代码。在编码扫描时, 扫描输出线不能直接用于键盘扫描, 而必须经过低电平有效输出的译码器。例如, 将 $SL_0 \sim SL_2$ 输入到通用的3-8译码器(74LS138)即可得到直接可用的扫描线(由8279内部逻辑所决定, 不能直接用4-16译码器对 $SL_0 \sim SL_3$ 进行译码, 即在编码扫描时, SL_3 仅用于显示

器，而不能用于键盘扫描)。

暂存于FIFO中的按键代码，在CPU执行中断处理子程序时取出，数据从FIFO取走后，中断请求信号IRQ将自动撤销。在中断子程序读取数据前，下一个键被按下，则该键代码自动进入FIFO。FIFO堆栈由8个8位的存储单元组成，允许依次暂存8个键的代码。这个栈的特点是先进先出，因此由中断子程序读取的代码顺序与键被按下的次序相一致。在FIFO中的暂存数据多于一个时，只有在读完（每读一个数据，则它从栈顶自动弹出）所有数据时，IRQ信号才会撤销。虽然键的代码暂存于8279的内部堆栈，但CPU从栈内读取数据时只能用“输入”或“取数”指令而不能用“弹出”指令，因为8279芯片在微机系统中是作为I/O接口电路而设置的。

在传感器扫描方式工作时，将对开关列阵中每一个结点的通、断状态（传感器状态）进行扫描，并且当列阵（最多是8×8位）中的任何一位发生状态变化时，便自动产生中断信号IRQ。此时，FIFO的8个存储单元用于寄存传感器的现时状态，称其为状态存储器。其中存储器的地址编号与扫描线的顺序一致。中断处理子程序将状态存储器的内容读入CPU，并与原有的状态比较后，便可由软件判断哪一个传感器的状态发生了变化。所以，8279用来检测开关（传感器）的通、断状态是非常方便的。

在选通输入方式工作时， $RL_0 \sim RL_7$ 与8255选通并行输入端口的功能完全一样。此时，CNTL端作为选通信号STB的输入端，STB为高电平有效。

此外，在使用8279时，不必考虑按键的抖动和串键问题。因在芯片内部已设置了消除触头抖动和串键的逻辑电路，这给使用带来了很大方便。

(2) 显示输出

8279内部设置了16×8显示数据存储器（RAM），每个单元寄存一个字符的8位显示代码。8个输出端与存储单元各位的对应关系为

D7	D6	D5	D4	D3	D2	D1	D0
A ₃	A ₂	A ₁	A ₀	B ₃	B ₂	B ₁	B ₀

A₃~A₀、B₃~B₀分时送出16个（或8个）单元内存储的数据，并在16个或8个显示器上显示出来。

显示器的扫描信号与键盘输入扫描信号是公用的，当实际的数码管显示器多于4个时，必须采用编码扫描输出，经过译码器后，方能用于显示器的扫描。

显示数据经过数据总线D7~D0及写信号 \overline{WR} （同时 $\overline{CS}=0$ ， C/\overline{D} （A0）=0）可以分别写入显示存储器的任何一个单元。一旦数据被写入后，8279的硬件便自动管理显示存储器的输出及同步扫描信号。因此，对操作者仅要求完成向显示存储器写入信息的操作。

8279的显示管理电路亦可在多种方式下工作，如左端输入、右端输入、8字符显示、16字符显示等。各种方式的设置将在后面加以说明。8279的工作方式是由各种控制命令决定的。CPU通过数据总线向芯片传送命令时，应使 $\overline{WR}=0$ 、 $\overline{CS}=0$ 及 C/\overline{D} （A0）=1。

8279共有8条命令，分述如下。

① 键盘、显示器工作模式设置命令。编码格式为

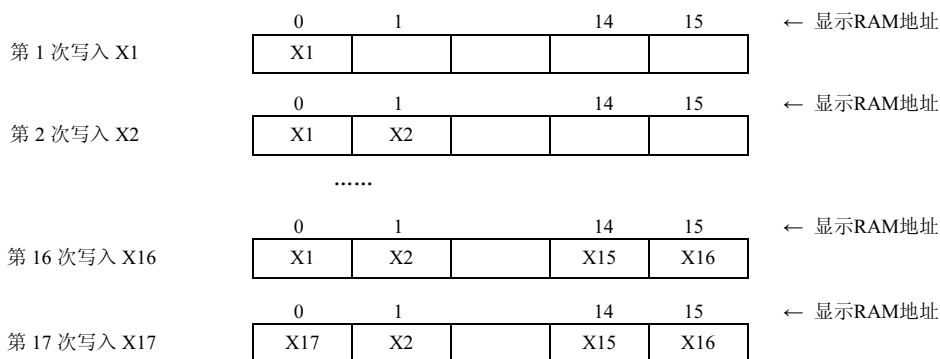
D7	D6	D5	D4	D3	D2	D1	D0
g0	0	0	D ₁	D ₀	K ₂	K ₁	K ₀

最高3位000是本命令的特征码（操作码）。D₁、D₀用于决定显示方式，定义为

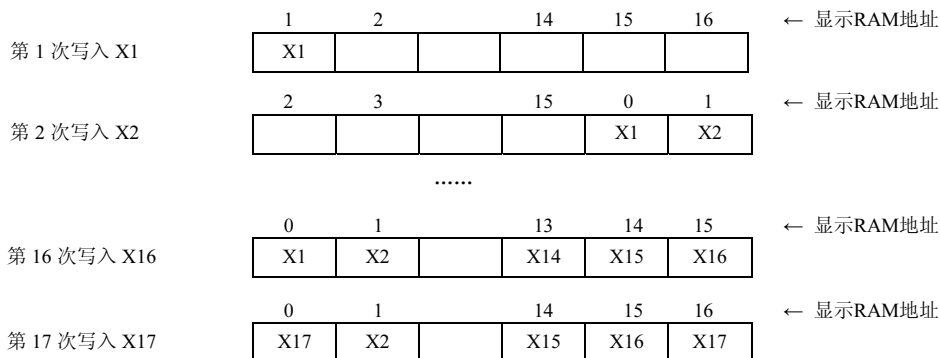
D ₁	D ₀	显示管理方式
0	0	8 字符显示; 左端输入
0	1	16 字符显示; 左端输入
1	0	8 字符显示; 右端输入
1	1	16 字符显示; 右端输入

8279可外接8位或16位的7段LED数码管显示器, 每一位显示器对应一个8位的显示RAM单元。显示RAM中的字符代码与扫描信号同步地依次送上输出线A₃~A₀、B₃~B₀。当实际的数码管显示器少于8时, 也必须设置8字符或16位字符显示模式之一。如果设置16字符显示, 则显示RAM中从“0”单元到“15”单元的内容同样依次轮流输出, 而不管扫描线上是否有数码管显示器存在。

左端输入方式是一种简单的显示模式。显示器的位置(最左边由SL₀驱动的显示器为“0”号位置)编号与显示RAM的地址一一对应, 即显示RAM中“0”地址的内容在“0”号(最左端)位置显示。CPU依次从“0”地址或某一地址开始将字符代码写入显示RAM。地址大于15时, 再从0地址开始写入。写入过程为



右端输入方式也是一种常用的显示方式, 一般的电子计算器都采用这种方式。从右端输入信号与前者比较的一个重要特点是显示RAM的地址与显示器的位置不是一一对应的, 而是每写入一个字符, 左移一位, 显示器最左端的内容被移出丢失。写入过程为



K₂、K₁、K₀用于设置键盘的工作方式, 定义为

K ₂	K ₁	K ₀	数据输入及扫描方式
0	0	0	编码扫描, 键盘输入, 两键互锁
0	0	1	译码扫描, 键盘输入, 两键互锁
0	1	0	编码扫描, 键盘输入, 多键有效
0	1	1	译码扫描, 键盘输入, 多键有效
1	0	0	编码扫描, 传感器阵列检测
1	0	1	译码扫描, 传感器阵列检测
1	1	0	选通输入, 编码扫描显示器
1	1	1	选通输入, 译码扫描显示器

在键盘扫描方式中, 两键互锁是指当被按下键未释放前, 第二键又被按下时, FIFO堆栈仅接收第一键的代码, 第二键作为无效键处理。如果两个键同时被按下, 则后释放的键为有效键, 而先释放者作为无效键处理。多键有效方式是指当多个键同时被按下, 则所有键依扫描顺序被识别, 其代码依次写入FIFO堆栈。虽然8279具有两种处理串键的方式, 但通常选用两键互锁方式, 以消除多余的被按下的键所带来的错误输入信息。

给8279加一个RESET信号将自动设置编码扫描, 键盘输入(两键互锁), 左端输入的16字符显示, 该信号的作用等效于编码为08H的命令。

② 扫描频率设置命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	P ₄	P ₃	P ₂	P ₁	P ₀

最高3位001是本命令的特征码。P₄P₃P₂P₁P₀取值为2~31, 是外接时钟的分频系数, 决定内部时钟频率。8279在接到RESET信号后, 如果不发送本命令, 则分频系数取值31。

③ 读FIFO堆栈的命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	AI	×	A ₂	A ₁	A ₀

最高3位010是本命令的特征码。在读FIFO之前, CPU必须先输出这条命令。8279接收到本命令后, CPU执行输入指令, 从FIFO中读取数据。地址由A₂A₁A₀决定, 如A₂A₁A₀=000, 则输入指令执行的结果是将FIFO堆栈顶(或传感器阵列状态存储器)的数据读入CPU的累加器。AI是自动增1标志, 当AI=1时, 每执行一次输入指令, 地址A₂A₁A₀自动加1。显然, 键盘输入数据时, 每次只需从栈顶读取数据, 故AI应取0。如果数据输入方式为检测传感器阵列的状态, 则AI应取1, 执行8次输入指令, 依次把FIFO的内容读入CPU。利用AI标志位可省去每次读取数据前都要设置读取地址的操作。

④ 读显示RAM命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	AI	A ₃	A ₂	A ₁	A ₀

最高3位011是本命令的特征码。在读显示RAM中的数据之前, 必须先输出这条命令, 8279接收到这条命令后, CPU才能读取数据。A₃A₂A₁A₀用于区别16个RAM地址, AI是地址自动加“1”标志。

⑤ 写显示RAM命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	A1	A ₃	A ₂	A ₁	A ₀

最高3位100是本命令的特征码。在将数据写入显示RAM之前，CPU必须先输出这条命令。命令中的地址码A₃A₂A₁A₀决定8279芯片接收来自CPU的数据存放在显示RAM的哪个单元。A1是地址自动增“1”标志。

⑥ 显示屏蔽消隐命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
1	0	1	×	IWA	IWB	BLA	BLB

最高3位101是本命令的特征码。IWA和IWB分别用以屏蔽A组和B组显示RAM。在双4位显示器使用时，即OUTA_{0~3}和OUTB_{0~3}独立地作为两个半字节输出时，可改写显示RAM中的低半字而不影响高半字节的状态（若IWA=1），或者可改写高半字节而不影响低半字节（若IWB=1）。BLA和BLB是消隐特征位，要消隐两组显示输出，必须使BLA和BLB同时为“1”，要恢复显示时，则使它们同时为“0”。

⑦ 清除命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
1	1	0	C _{D2}	C _{D1}	C _{D0}	C _F	C _A

最高3位110是本命令的特征码。C_{D2}、C_{D1}、C_{D0}用来设定清除显示RAM的方式，定义为

C _{D2}	C _{D1}	C _{D0}	清除方式
1	0	×	显示RAM所有单元均置“0”
	1	0	显示RAM所有单元均置“20H”
	1	1	显示RAM所有单元均置“1”
0	×	×	不清除（C _A =0时）

C_F=1，清除FIFO状态标志，FIFO被置成空状态（无数据），并复位中断输出IRQ。

C_A是总清的特征位，C_A=1，清除FIFO状态和显示RAM（方式仍由C_{D1}、C_{D0}确定）。

清除显示RAM大约需160μs，在此期间，CPU不能向显示RAM写入数据。

⑧ 中断结束/设置出错方式命令。编码格式为

D7	D6	D5	D4	D3	D2	D1	D0
1	1	1	E	×	×	×	×

最高3位111是本命令的特征码。在传感器工作方式中，该命令使IRQ输出线变为低电平（即中断结束），允许再次对RAM写入（在检测到传感器变化后，IRQ可能已经变成高电平，这时禁止在复位前再次将信息写入RAM）。在N键巡回工作方式中，若E=1，则在消颤期内，当有多键同时被按下时，产生中断，并且阻止对RAM的写入。

除了上述8条命令之外，8279还有一个状态字。状态字用来指出FIFO中的字符个数、出错信息及能否对显示RAM进入写入操作。状态字格式为

D7							D0
DU	S/E	O	U	F	N ₂	N ₁	N ₀

N₂N₁N₀表示FIFO中数据的个数。

F=1时，表示FIFO已满（存有8个键入数据）。

在FIFO中没有输入字符时，CPU读FIFO，则置“1”U。

当FIFO已满，又输入一个字符时发生溢出，置“1”O

S/E用于传感器扫描方式，几个传感器同时闭合时置“1”。

在清除命令执行期间，DU为“1”，此时对显示RAM写操作无效。

6.3.3 8279接口应用编程

图6.18为单片机8051与8279组成的键盘显示器接口电路。8051的P2.7（A15）接到8279的片选端CS，P2.0（A8）接到8279的C/D（A0）端。因此，该接口对用户来说只有两个口地址：命令口地址7FFFH和数据口地址7EFFH。图中，8279外接4×4矩阵键盘和6位共阴极LED数码管，采用编码扫描方式，译码器74LS138对扫描线SL₀~SL₃进行译码，译码输出一方面扫描矩阵键盘，另一方面同时又作为LED数码管的位驱动。

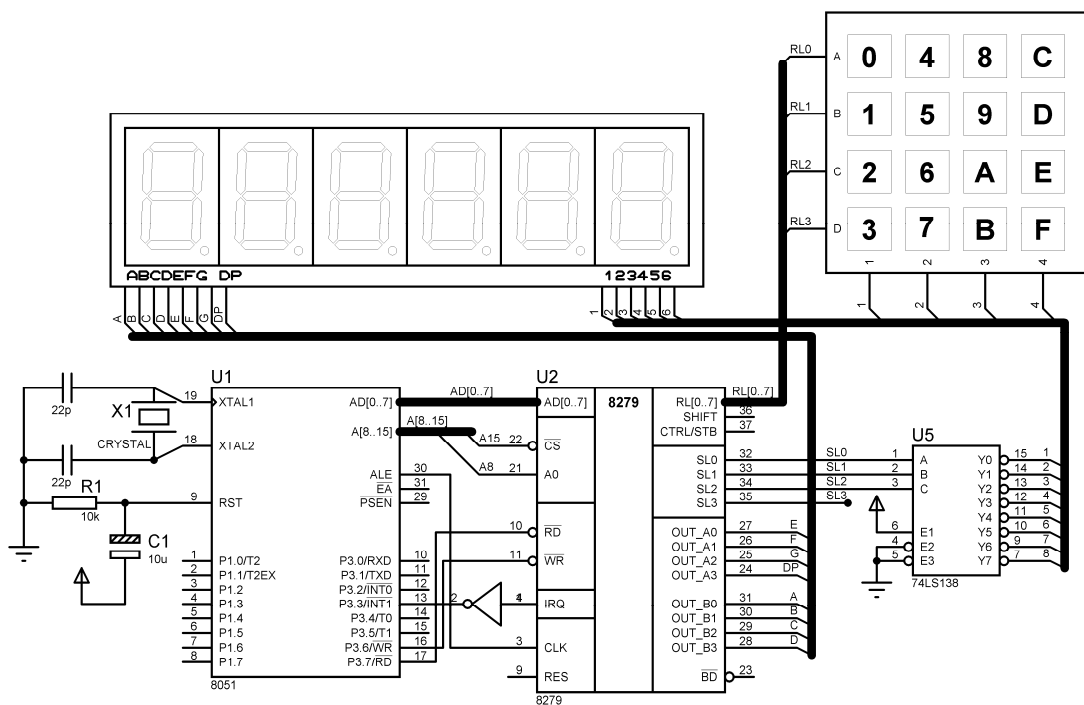


图6.18 单片机8051与8279组成的键盘显示器接口电路

例6-8为采用C51编写的8279应用程序。主程序先在8051单片机内部RAM中开辟一段显示缓冲区，并将显示字符写入其中，接着调用8279的初始化子程序，根据需要对8279进行初始化并开中断，然后不断循环调用8279显示更新子程序，将显示缓冲区中的内容显示到数码管上。有按键压下时将触发8051中断，通过8279中断服务程序读取键值，并将键值送入显示缓冲区。程序执行后，数码管将显示“012345”字符，有键被按下时，相应的键值将显示在第一个数码管上。

例6-8 采用C51编写的8279应用程序。

```

#include <absacc.h>
#define uchar unsigned char
#define uint unsigned int
char data DisBuf[6]={0,1,2,3,4,5};           //显示缓冲区
uchar code keyval[]={0x00,0x01,0x02,0x03,0x08,0x09,0x0a,0x0b, //键值表
                    0x10,0x11,0x12,0x13,0x18,0x19,0x1a,0x1b};
uchar code SEG[]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07, //段码表
                  0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71,0x00};

/***** 8279初始化函数 *****/
void KbDisInit() {
    XBYTE[0x7fff]=0x00;           //设置8279工作方式
    XBYTE[0x7fff]=0xD1;           //清除8279
    while (XBYTE[0x7fff] & 0x80); //等待清除结束
    XBYTE[0x7eff]=0x34;           //设置8279分频系数
}

/***** 读键值函数 *****/
uchar ReadKey(){
    uchar i,j;
    if (XBYTE[0x7fff] & 0x07){    //判断是否有按键
        XBYTE[0x7fff]=0x40;       //有键按下，写入读FIFO命令
        i=XBYTE[0x7eff];          //获取键值
        j=0;
        while (i!=keyval[j]){j++;} //查键值表
        return(j+1);
    }
    return (0);                  //无键按下
}

/***** 显示函数 *****/
void Disp() {
    uchar i;
    XBYTE[0x7fff]=0x90;           //写显示RAM命令
    for (i=0; i<6; i++){
        XBYTE[0x7eff]=SEG[DisBuf[i]]; //显示缓冲区内容
    }
}

/***** 填充显示缓冲区函数 *****/
void DspBf(){
    uchar i;
    for (i=1; i<6; i++){
        DisBuf[i]=0x10;
    }
}

/***** 无按键处理函数 *****/
void NoKey() {

```

```

;
}

/***** 0键处理函数 *****/
void k0() {
    DisBuf[0]=0x00; DspBf();
}

/***** 1键处理函数 *****/
void k1() {
    DisBuf[0]=0x01; DspBf();
}

/***** 2键处理函数 *****/
void k2() {
    DisBuf[0]=0x02; DspBf();
}

/***** 3键处理函数 *****/
void k3() {
    DisBuf[0]=0x03; DspBf();
}

/***** 4键处理函数 *****/
void k4() {
    DisBuf[0]=0x04; DspBf();
}

/* k5, ...其他按键处理函数可在此处插入 */

code void (code * KeyProcTab[]) ()={NoKey, k0,k1,k2,k3,k4,k5/*... */};

/***** 主函数 *****/
void main(){
    KbDisInit();    //8279初始化
    while(1){
        Disp();
        (* KeyProcTab[ReadKey()]) ();    //根据不同按键的值查表散转
    }
}

```

6.4 点阵字符型LCD接口技术

液晶显示器LCD体积小、重量轻、功耗低，应用十分广泛，是一种被动式显示器。它本身并不发光，只是调节光的亮度。目前，常用的LCD是根据液晶的扭曲一向列效应原理制成的。这是一种电场效应，夹在两块导电玻璃电极之间的液晶经过一定的处理后，其内部的分子呈90°的扭曲。这种液晶具有旋光特性。当线性偏振光通过液晶层时，偏振面会旋转90°。

当给玻璃电极加上电压后，在电场的作用下，液晶的扭曲结构消失，其旋光作用也随之消失，偏振光便可以直接通过。当去掉电场后，液晶分子又恢复其扭曲结构。把这样的液晶放在两个偏振片之间，改变偏振片的相对位置（平行或正交），就可得到黑底白字或白底黑字的显示形式。

LCD常采用交流驱动，通常采用异或门把显示控制信号和显示频率信号合并为交变的驱动信号，如图6.19所示。当显示控制电极上的波形与公共电极上的方波相位相反时，则为显示状态。显示控制信号由C端输入，高电平为显示状态。显示频率信号是一个方波。当异或门的C端为低电平时，输出端B的电位与A端相同，LCD两端的电压为0，LCD不显示，当异或门的C端为高电平时，B端的电位与A端相反，LCD两端呈现交替变化的电压，LCD显示。常用的扭曲一向列型LCD，其驱动电压范围为3~6V。由于LCD是容性负载，因此工作频率越高，消耗的功率越大，而且显示频率升高，对比度会变差，当频率升高到临界高频以上时，LCD就不能显示了，所以LCD宜采用低频工作。

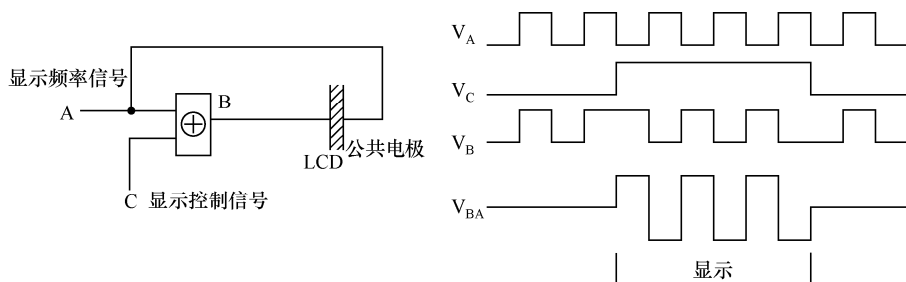


图6.19 LCD的基本驱动电路及波形

6.4.1 点阵字符型LCD显示模块

点阵字符型LCD显示模块能显示数字、字母、符号及少量自定义图形符号，如简单的汉字等，因而在单片机应用系统中得到了广泛应用。点阵字符型LCD显示模块由LCD显示器、点阵驱动器、LCD控制器等组成。模块内集成有字符发生器和数据存储器，采用单一+5V电源供电。点阵字符型液晶显示模块在国际上已经规范化，所采用的控制器多为日立公司的HD44780，也有采用其兼容电路，如SED1278（SEIKO、EPSON公司产品）、KS0666（三星公司产品）等。表6.15列出了EPSON公司生产的EA-D系列各型号点阵字符型LCD显示模块的外部特性。

表6.15 EPSON公司生产的EA-D系列各型号点阵字符型LCD显示模块的外部特性

名称	字符数	外部尺寸	视觉范围	字符点阵	字符尺寸	点的尺寸	速率
EA-D16015	16×1	80×36	64.5×13.8	5×7	3.07×6.56	0.55×0.75	1/16
EA-D16025	16×2	84×44	61×15.8	5×7	2.96×5.56	0.56×0.66	1/16
EA-D20025	20×2	116×37	83×18.6	5×7	3.20×5.55	0.60×0.65	1/16
EA-D20040	20×4	98×60	76×25.2	5×7	3.01×4.84	0.57×0.57	1/16
EA-D24016	24×1	126×36	100.0×13.8	5×10	3.15×8.70	0.55×0.70	1/11
EA-D40016	40×1	182×33.5	154.4×15.8	5×10	3.15×8.70	0.55×0.70	1/11
EA-D40025	40×2	182×33.5	154.4×15.8	5×7	3.20×5.55	0.60×0.65	1/16

下面介绍EPSON公司生产的点阵字符型LCD显示模块与单片机系统的接口及应用。EA-D系列点阵字符型液晶显示模块内的部结构如图6.20所示。它由点阵式液晶显示面板、SED1278控制器及4个列驱动器组成。SED1278完成显示模块的时序控制，同时也可以驱动16行、40列的点阵库。

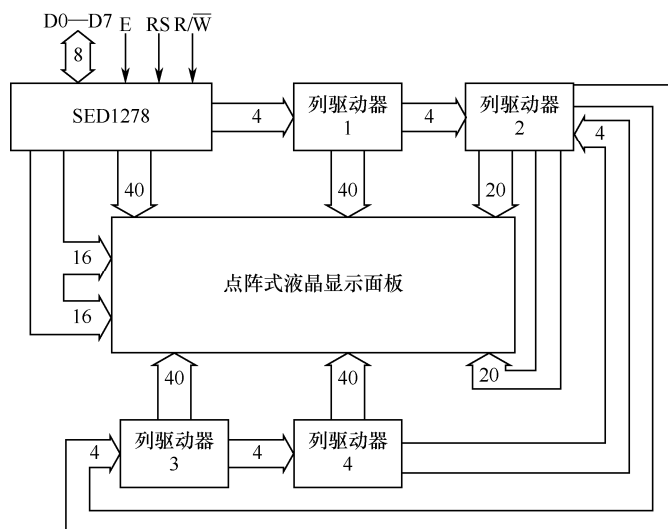


图6.20 EA-D系列点阵字符型液晶显示模块的内部结构

SED1278控制器有14条引脚：

V_{SS} ：地线输入端。

V_{DD} ：+5V电源输入端。

V_O ：液晶显示面板亮度调节，通过10~20k Ω 的电阻接到+5V和地之间起调节显示亮度的作用。

RS：寄存器选择信号输入线，低电平选通指令寄存器，高电平选通数据寄存器。

R/\overline{W} ：读/写信号输入线，低电平为写入，高电平为读出。

E：片选信号输入线，高电平有效。

$D_0 \sim D_7$ ：数据总线，可以选择4位总线或8位总线操作，选择4位总线操作时使用 $D_4 \sim D_7$ 。

SED1278的控制电路主要由指令寄存器（IR）、数据寄存器（DR）、忙标志（BF）、地址计数器（AC）、显示数据寄存器（DDRAM）、字符发生器ROM（CGROM）、字符发生器RAM（CGRAM）、时序发生电路所组成。

指令寄存器IR用于寄存各种指令码，只能写入，不能读出。

数据寄存器DR用于寄存显示数据，由内部操作自动写入DDRAM和CGRAM，或寄存从DDRAM和CGRAM读出的数据。

忙标志BF=1时，表示正在进行内部操作，此时不能接受任何外部指令和数据。

地址计数器AC作为DDRAM或CGRAM的地址指针。如果地址码随指令写入IR，则IR的地址码自动装入AC，同时选择DDRAM或CGRAM单元。

显示数据寄存器DDRAM用于存储显示数据，DDRAM的地址与显示屏幕的物理位置是一一对应的，当向数据寄存器某一地址单元写入一个字符的编码时，该字符就在对应的位置

上显示出来。表6.16列出了DDRAM显示地址与显示屏物理位置的对应关系。

表6.16 DDRAM显示地址与显示屏物理位置的对应关系

显示地址 行号	列号																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53
3	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
4	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67

字符发生器CGROM由8位字符码生成5×7点阵字符160个和5×10点阵字符32个，已经固化在LCD显示器模块内部，由用户随意使用。表6.17列出了8位字符编码的高、低位排列及其与字符的对应关系。如果想显示192个字符中的一个，则只要把该字符的编码送入DDRAM即可。如果想显示192个字符以外的字符，则需要利用CGRAM自定义字符。

表6.17 8位字符编码的高、低位排列及其与字符的对应关系

高位 低位	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
0000		0	@	P	\	p		-	々	ミ	α	p
0001	!	1	A	Q	a	q	.	ヌ	チ	ム	a	q
0010	“	2	B	R	b	r	「	イ	ツ	メ	β	θ
0011	#	3	C	S	c	s	」	ウ	チ	モ	ε	∞
0100	\$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
0101	%	5	E	U	e	u	。	オ	ナ	ユ	σ	O
0110	&	6	F	V	f	v	ラ	カ	ニ	ヨ	ρ	Σ
0111	,	7	G	W	g	w	ア	キ	ヌ	ラ	g	π
1000	(8	H	X	h	x	イ	ク	ネ	リ	∫	X
1001)	9	I	Y	i	y	ウ	ケ	ノ	ル	-1	Y
1010	*	:	J	Z	j	z	エ	コ	ハ	レ	j	千
1011	+	:	K	[k	{	オ	サ	ヒ	ロ	×	万
1100	,	<	L	¥	l		セ	シ	フ	ワ	Φ	⊕
1101	-	=	M]	m	}	コ	ス	ヘ	ン	£	÷
1110	.	>	N	^	n	→	ヨ	セ	ホ	ハ	n	
1111	/	?	O	-	o	←	ツ	ソ	マ	ロ	○	■

字符发生器CGRAM是为用户创建自己的特殊字符设立的，容量为64字节，地址为00~3FH，但是作为自定义字符使用的仅是一个字节中的低5位，每个字节的高3位可作为数据存储器使用。若自定义字符为5×7点阵，则可定义8个字符。若自定义字符为5×10点阵，则可定义4个字符。自定义字符的编码为00H~07H。表6.18列出了自定义字符“上”。从表中可以看出，字符编码（DDRAM中的数据）的0~2位等同于CGRAM地址的3~5位。CGRAM地址的0~2位定义字符的行位置。CGRAM中数据的0~4位决定字符形式，第4位是字符的最左端。CGRAM的5~7位不用作显示字符，因此可用作一般的数据RAM。

表6.18 自定义字符“上”

字符编码 (DDRAM数据)								CGRAM地址						字符形式 (CGRAM数据)							
7	6	5	4	3	2	1	0	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	0	0	×	0	0	0	0	0	0	0	0	0	×	×	×	0	0	1	0	0
														×	×	×	0	0	1	0	0
														×	×	×	0	0	1	0	0
														×	×	×	0	0	1	1	1
														×	×	×	0	0	1	0	0
														×	×	×	0	0	1	0	0
														×	×	×	1	1	1	1	1
														×	×	×	0	0	0	0	0

点阵字符型LCD显示模块的显示功能是由各种命令来实现的，共有11条命令。

① 清显示命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1

该命令把空格编码20H写入显示数据存储器的所有单元。

② 光标返回命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	1	×

该命令把地址计数器中DDRAM地址清0，如果显示屏上显示了字符，则光标移到起始位置。如果显示两行，则光标移到第一行第一个字符的位置，显示数据存储器的内容不变。

③ 设置输入方式命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	1	I/D	S

当一个字符编码被写入DDRAM或从DDRAM中读出时，若I/D=1，则DDRAM地址加1，若I/D=0，则DDRAM地址减1。地址加1时，光标右移；地址减1时，光标左移。对CGRAM的读/写操作与DDRAM一样，只是CGRAM与光标无关。当S=1时，整个显示屏向左（I/D=1）或向右（I/D=0）移动。在从DDRAM中读数、向CGRAM写数或从CGRAM中读数、S=0这三种情况下，显示屏不移动。

④ 显示开/关控制命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	1	D	C	B

当D=0时，显示器关闭，显示数据存储器的数据不变；当D=1时，显示器立即显示DDRAM中的数据。

当C=0时，不显示光标；当C=1时，显示光标。当选择字符为5×7点阵时，用第8行的第5个点显示光标。

当B=1时，显示闪烁光标，当时钟为270kHz时，在379.2ms内交换显示全黑点和字符，以实现字符闪烁。

⑤ 光标或显示屏移动命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	1	S/C	R/L	×	×

该命令使显示和光标向左或向右移位。对两行显示而言，光标从第一行的第40 个字符位置移到第二行的首位。从第二行的第40个位置不能移位到清屏的起始位置，而是回到第二行的第一个位置。命令中S/C和R/L位的作用如下：

S/C	R/L	作 用
0	0	光标左移，地址计数器减 1
0	1	光标右移，地址计数器加 1
1	0	显示屏左移，光标跟随显示屏移动
1	1	显示屏右移，光标跟随显示屏移动

⑥ 功能设置命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	IF	N	F	×	×

命令中的IF位用来设置接口数据长度，当IF=1时，数据以8位长度（D7～D0）发送或接收，当IF=0时，数据以4位长度（D7～D4）发送或接收。命令中的N和F位用来设置显示屏的行数和字符的点阵。设置方式为

N	F	显示行数	字符点阵	占空系数
0	0	1	5×7	1/16
0	1	1	5×10	1/11
1	0	2	5×7	1/16
1	1	2	5×7	1/16

对于EA-D20040来说，一定要设置N=1，显示2行。

⑦ 设置CGRAM地址命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	A5	A4	A3	A2	A1	A0

该命令的功能是设置CGRAM的地址，命令执行后，单片机和CGRAM可连续进行数据交换。

⑧ 设置DDRAM地址命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	A6	A5	A4	A3	A2	A1	A0

该命令的功能是设置DDRAM的地址，命令执行后，单片机与DDRAM进行数据交换。

⑨ 读忙标志和地址命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	1	BF	A6	A5	A4	A3	A2	A1	A0

该命令的功能是读出忙标志BF的值。当读出的BF=1时，说明系统内部正在进行操作，不能接收下一条命令。在读出BF值的同时，CGRAM和DDRAM 所使用地址计数器的值也被同时读出。

⑩ 向CGRAM或DDRAM写数据命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	0	D	D	D	D	D	D	D	D

该命令的功能是把二进制数DDDDDDDD写入CGRAM或DDRAM中，若先送入CGRAM的地址，则向CGRAM写入；若先送入DDRAM的地址，则向DDRAM写入。

⑪ 从CGRAM或DDRAM读取数据命令。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	1	D	D	D	D	D	D	D	D

该命令的功能是将数据从写数据命令建立的CGRAM或DDRAM地址指出的RAM中读出。在本命令之前的命令应是CGRAM或DDRAM地址建立命令、光标移位命令或是上次CGRAM/DDRAM数据读出命令。若是其他命令，则读出的数据可能会出错。

在执行读数据或写数据命令之后，地址计数器会自动加1或减1。一般是先执行一条地址建立命令或光标移位命令，再执行读数据命令，一旦一条读数据命令被执行后，就可连续执行数据读取命令，而不需再执行其他命令了。

6.4.2 直接方式接口应用编程

图6.21为16字符×2行的点阵字符型LCD显示模块与单片机8051的直接方式接口电路。LCD显示模块的R/W和RS信号由8051单片机的低8位地址线来控制，显示模块的E信号则由单片机的最高地址线P2.7和读RD、写WR信号线组成的联合逻辑电路来控制，从而可得该接口电路的命令写入地址为7FF0H，命令读取地址为7FF1H，数据操作地址为7FF2H。这种接口被称为直接方式接口。

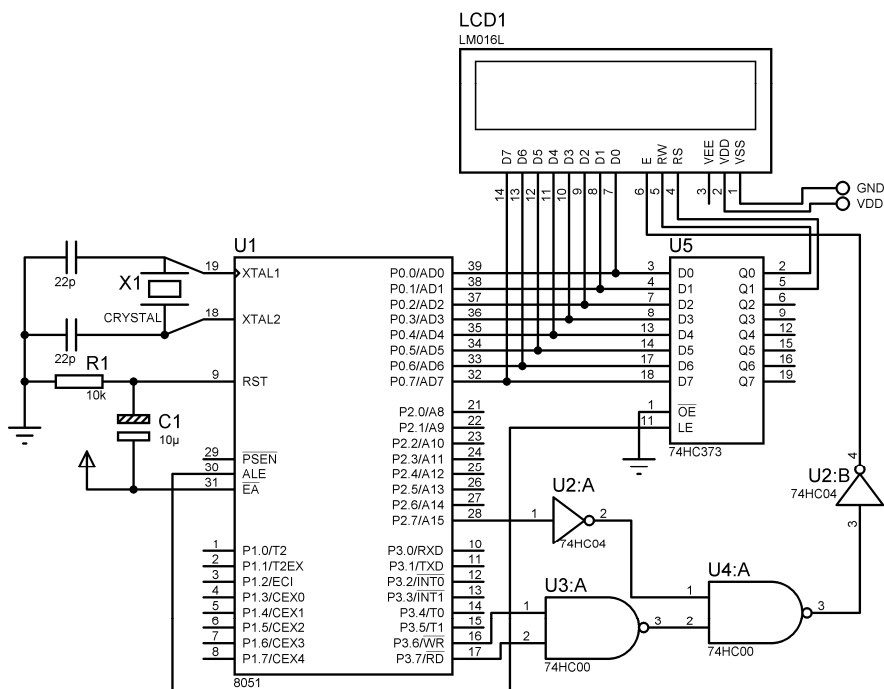


图6.21 16字符×2行的点阵字符型LCD显示模块与单片机8051的直接方式接口电路

例6-9列出了直接方式接口的C51驱动程序。进入主程序后，首先调用LCD模块初始化子程序，初始化内容包括将功能设置（8位字长、2行、5×7点阵）、清屏、设置输入方式和设置显示方式及光标等。需要注意的是，每写入一条命令，都应先检查忙标志BF，只有当BF=0时才能执行下一条指令。接着调用自定义汉字字符子程序，在该子程序中先设定CGRAM首地址，然后依次向CGRAM中写入各个自定义汉字的字模数据，接着设定显示字符在液晶屏上的位置，即DDRAM的地址，最后将要显示的字符代码分别写入DDRAM，对于CGROM中的字符代码可以通过查表6.17得到，而自定义汉字字符的代码则为00H~07H，本例只定义了3个字符“年”、“月”、“日”，它们的代码分别为00H、01H和02H。

例6-9 点阵字符型LCD模块直接方式接口的C51驱动程序。

```
#include <reg51.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int
#define Busy    0x80                                // 忙判别位

char xdata Lcd1602CmdPort _at_ 0x7ff0;               //LCD命令口地址
char xdata Lcd1602StatusPort _at_ 0x7ff1;           //LCD状态口地址
char xdata Lcd1602WdataPort _at_ 0x7ff2;           //LCD数据口地址

code char exampl[]="Hello Every Body";
code char examp2[]={0x32,0x30,0x31,0x35,0x00,0x35,0x01,0x32,0x36,0x02};
code char Hzzimo[]={0x08,0x0F,0x12,0x0F,0x0A,0x1F,0x02,0x00, //“年”
                    0x0F,0x09,0x0F,0x09,0x0F,0x09,0x11,0x00, //“月”
                    0x0F,0x09,0x09,0x0F,0x09,0x09,0x0F,0x00}; //“日”

/***** 写控制字符函数 *****/
void LcdWriteCommand( uchar CMD,uchar AttribC ){
    if (AttribC) while( Lcd1602StatusPort & Busy );    // 检测忙信号
    Lcd1602CmdPort = CMD;
}

/***** 当前位置写字符函数 *****/
void LcdWriteData( char dataW ) {
    while( Lcd1602StatusPort & Busy );                // 检测忙信号
    Lcd1602WdataPort = dataW;
}

/***** 显示光标定位函数 *****/
void LocateXY( char posx,char posy) {
    uchar temp;
    temp = posx & 0xf;
    posy &= 0x1;
    if ( posy )temp |= 0x40;
    temp |= 0x80;
    LcdWriteCommand(temp,0);
}

/***** 自定义汉字字符函数 *****/
```

```

void Hz(){
    uchar i;
    LcdWriteCommand( 0x40,1 );
    for (i=0;i<24;i++){
        LcdWriteData(Hzzimo[i]);
    }
}

/***** 单字符显示函数 *****/
void DispOneChar(uchar x,uchar y,uchar Wdata) {
    LocateXY( x, y );           // 定位显示字符的x,y位置
    LcdWriteData( Wdata );       // 写字符
}

/***** 显示字符串函数 *****/
void ePutstr(uchar x,uchar y, uchar j,uchar code *ptr){
    uchar i,l=0;
    for (i=0;i<j;i++){
        DispOneChar(x++,y,ptr[i]);
        if ( x == 16 ){
            x = 0; y ^= 1;
        }
    }
}

/***** 5ms延时函数 *****/
void Delay5ms(void){
    uint i = 5552;
    while(i--);
}

/***** 400ms延时函数 *****/
void Delay400ms(void){
    uchar i = 5;
    uint j;
    while(i--){
        j=7269;
        while(j--);
    };
}

/***** LCD初始化函数 *****/
void LcdReset( void ){
    LcdWriteCommand( 0x38, 0);           // 显示模式设置 (不检测忙信号)
    Delay5ms();
    LcdWriteCommand( 0x38, 0);           // 共三次
    Delay5ms();
    LcdWriteCommand( 0x38, 0);
    Delay5ms();
    LcdWriteCommand( 0x38, 1);           // 显示模式设置 (以后均检测忙信号)
    LcdWriteCommand( 0x08, 1);           // 显示关闭
    LcdWriteCommand( 0x01, 1);           // 显示清屏
}

```

```

        LcdWriteCommand( 0x06, 1);          // 显示光标移动设置
        LcdWriteCommand( 0x0c, 1);          // 显示开及光标设置
    }

    /***** 主函数 *****/
    void main(void){
        uchar temp;
        Delay400ms();                        // 启动时必须的延时, 等待LCD进入工作状态
        LcdReset();                          // LCD初始化
        temp = 32;
        Hz();
        ePutstr(0,0,16,exempl);              // 第一行从第0位开始显示Hello Every Body
        ePutstr(4,1,10,examp2);              // 第二行从第4位开始显示2015年5月26日
        while(1);
    }

```

6.4.3 间接方式接口应用编程

点阵字符型LCD显示模块还可以采用间接控制方式与单片机进行接口。这种方式通过单片机的并行I/O端口引脚实现对LCD显示模块的间接控制。

图6.22给出了点阵字符型LCD显示模块与单片机的间接方式接口电路。LCD显示模块的RS、R/W和E信号分别由8051单片机的P2.1、P2.2和P2.3来控制。与直接方式不同, 间接控制

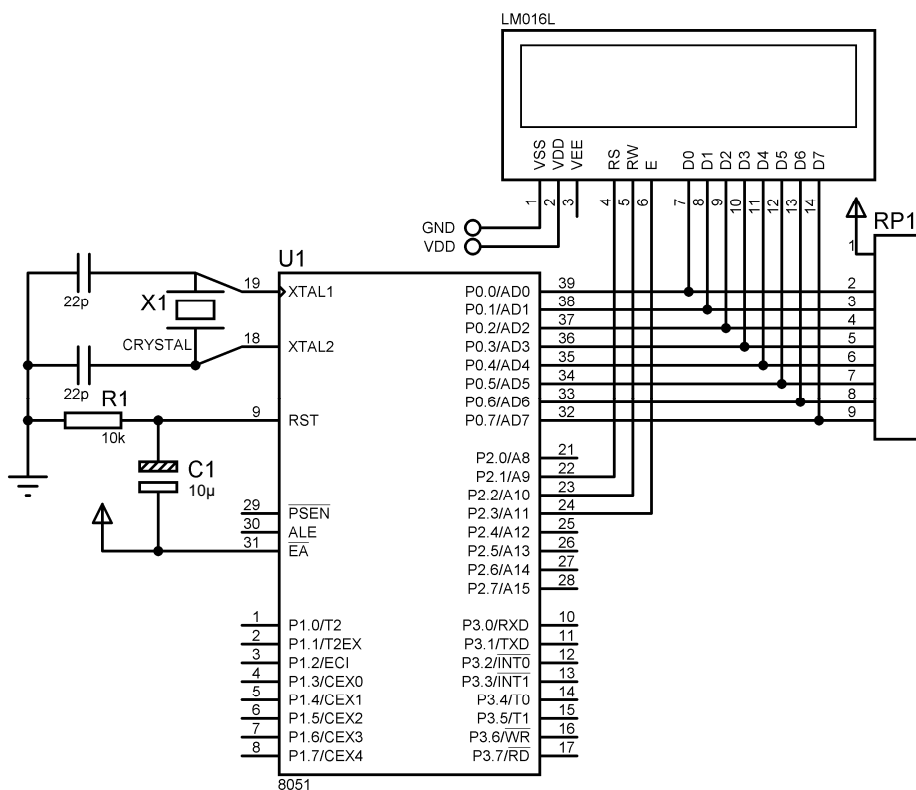


图6.22 点阵字符型LCD显示模块与单片机的间接方式接口电路

方式不是通过固定的接口地址，而是通过单片机I/O端口引脚来操作LCD显示模块，因此在编写驱动程序时要注意时序的配合。写操作时，E信号的下降沿有效，工作时序上应先设置RS、R/W状态，再写入数据，然后产生E信号脉冲，最后复位RS、R/W状态。读操作时，E信号的高电平有效，工作时序上应先设置RS、R/W状态，设置E信号为高电平，再读取数据，然后将E信号设置为低电平，最后复位RS、R/W状态，程序编写时要特别注意，工作时序的配合。例6-10列出了点阵字符型LCD模块间接方式接口的C51驱动程序。

例6-10 点阵字符型LCD模块间接方式接口的C51驱动程序。

```
#include <reg51.h>
#include<intrins.h>
#include<stdio.h>
#include<string.h>
#include<math.h>
#include<absacc.h>
#define uchar unsigned char
#define uint unsigned int
#define DataPort P0           // 数据端口
#define Busy    0x80

sbit    RS   = P2^1;           //LCD控制引脚定义
sbit    RW   = P2^2;
sbit    Elcm  = P2^3;

code char exampl[]="Hello Every Body";
unsigned char tem1,t;
unsigned char c1=10;

/***** 1Ms延时函数 *****/
void Delay1ms(void){
    uint i = 552;
    while(i--);
}

/***** 5Ms延时函数 *****/
void Delay5ms(void){
    uint i = 5552;
    while(i--);
}

/***** 等待允许函数 *****/
void WaitForEnable( void ) {
    DataPort = 0xff;
    RS =0; RW = 1; _nop_();
    Delay1ms();
}
```

```

        Elcm = 1; _nop_(); _nop_();
        Delay1ms();
        while( DataPort & Busy );
        Elcm = 0;
    }

    /***** 写控制字符函数 *****/
void LcdWriteCommand( uchar CMD,uchar AttribC ) {
    if (AttribC) WaitForEnable();           // 检测忙信号?
    RS = 0; RW = 0; _nop_();
    DataPort = CMD; _nop_();                // 送控制字子程序
    Elcm = 1;_nop_();_nop_();Elcm = 0;      // 操作允许脉冲信号
}

    /***** 当前位置写字符函数 *****/
void LcdWriteData( char dataW ) {
    WaitForEnable();                       // 检测忙信号
    RS = 1; RW = 0; _nop_();
    DataPort = dataW; _nop_();
    Elcm = 1; _nop_(); _nop_(); Elcm = 0;   // 操作允许脉冲信号
}

    /***** 显示光标定位函数 *****/
void LocateXY( char posx,char posy) {
    uchar temp;
    temp = posx & 0xf;
    posy &= 0x1;
    if ( posy )temp |= 0x40;
    temp |= 0x80;
    LcdWriteCommand(temp,0);
}

    /***** 单字符显示函数 *****/
void DispOneChar(uchar x,uchar y,uchar Wdata) {
    LocateXY( x, y );                     // 定位显示字符的x,y位置
    LcdWriteData( Wdata );                 // 写字符
}

    /***** 显示字符串函数 *****/
void ePutstr(uchar x,uchar y,uchar j, uchar code *ptr){
    uchar i;
    for (i=0;i<j;i++) {
        DispOneChar(x++,y,ptr[i]);
        if ( x == 16 ){
            x = 0; y ^= 1;

```



```

    }
}

/***** LCD初始化函数 *****/
void LcdReset( void ) {
    LcdWriteCommand( 0x38, 0);          // 显示模式设置 (不检测忙信号)
    Delay5ms();
    LcdWriteCommand( 0x38, 0);          // 共三次
    Delay5ms();
    LcdWriteCommand( 0x38, 0);
    Delay5ms();
    LcdWriteCommand( 0x38, 1);          // 显示模式设置 (以后均检测忙信号)
    LcdWriteCommand( 0x08, 1);          // 显示关闭
    LcdWriteCommand( 0x01, 1);          // 显示清屏
    LcdWriteCommand( 0x06, 1);          // 显示光标移动设置
    LcdWriteCommand( 0x0c, 1);          // 显示开及光标设置
}

/***** 400ms延时函数 *****/
void Delay400ms(void) {
    uchar i = 5;
    uint j;
    while(i--){
        j=7269;
        while(j--);
    };
}

/***** 主函数 *****/
void main(void){
    LcdReset();
    Delay400ms();
    ePutstr(0,0,16,exempl);           //第一行从第0位开始显示Hello Every Body
    while(1);
}

```

6.4.4 4位数据总线接口应用编程

点阵字符型LCD显示模块以间接方式与单片机进行接口时，为了节省单片机的I/O端口，还可以采用4位数据总线，接口电路如图6.23所示。LCD显示模块的RS、RW和E信号分别由单片机的P2.0、P2.1和P2.2控制，由于采用了4位数据总线，需要分两次向LCD传递数据。例6-11列出了点阵字符型LCD显示模块与单片机4位数据总线接口的C51驱动程序。

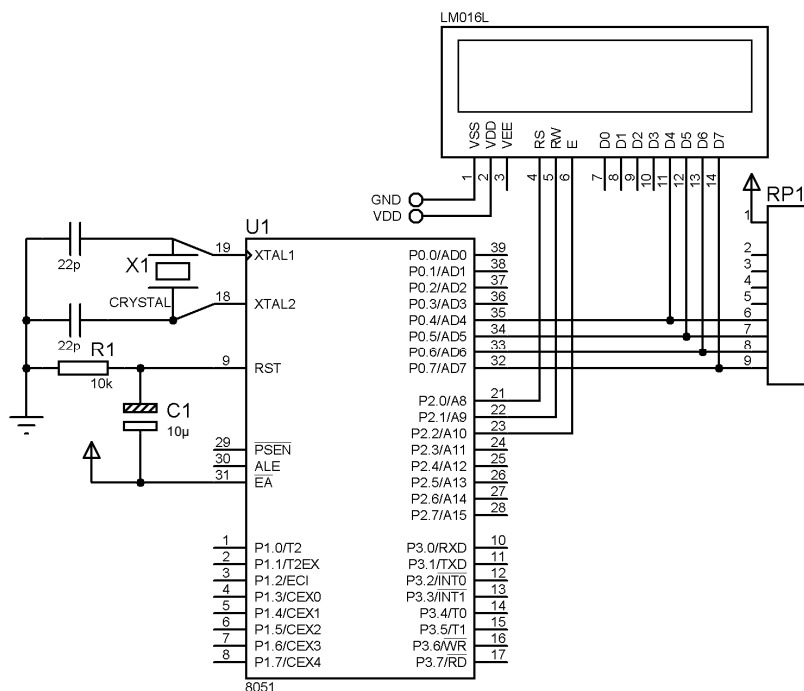


图6.23 点阵字符型LCD显示模块与单片机的4位数据总线接口

例6-11 点阵字符型LCD显示模块4位数据总线方式接口的C51驱动程序。

```
#include <STC15F2K.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int

sbit RS = P2^0;
sbit RW = P2^1;
sbit E = P2^2;

code char exampl[]={0x32,0x30,0x31,0x35,0x00,0x35,0x01,0x32,0x36,0x02};
code char Hzzimo[]={0x08,0x0F,0x12,0x0F,0x0A,0x1F,0x02,0x00, //“年”
                    0x0F,0x09,0x0F,0x09,0x0F,0x09,0x11,0x00, //“月”
                    0x0F,0x09,0x09,0x0F,0x09,0x09,0x0F,0x00}; //“日”

/***** 毫秒延时函数 *****/
void Delaysms() {
    uchar i, j;
    i = 12; j = 169;
    do{
        while (--j);
    } while (--i);
}
```

```

/***** 延时函数 *****/
void Delay(uchar t){
    while(--t) Delays();
}

/***** 写命令函数 *****/
void WriteCommand(uchar c){
    RS=0; RW=0;          //写指令
    E=1; P0=c; Delay(3); //写低4位
    E=0; _nop_();
    E=1; P0=c<<4; Delay(3); //写高4位
    E=0;
}

/***** 写数据函数 *****/
void WriteData(uchar c){
    RS=1; RW=0;          //写数据
    E=1; P0=c; Delay(3); //写低4位
    E=0; _nop_();
    E=1; P0=c<<4; Delay(3); //写高4位
    E=0;
}

/***** 显示光标定位函数 *****/
void LocateXY( char posx,char posy) {
    uchar temp;
    temp = posx & 0xf;
    posy &= 0x1;
    if ( posy )temp |= 0x40;
    temp |= 0x80;
    WriteCommand(temp);
}

/***** 显示单个字符函数 *****/
void DispOneChar(uchar x,uchar y,uchar Wdata) {
    LocateXY( x, y );          //定位显示字符的x,y位置
    WriteData( Wdata );        //写字符
}

/***** 显示字符串函数 *****/
void ePutstr(uchar x,uchar y,uchar j, uchar code *ptr){
    uchar i;
    for (i=0;i<j;i++) {
        DispOneChar(x++,y,ptr[i]);
        if ( x == 16 ){
            x = 0; y ^= 1;

```

```

    }
}

/***** LCD初始化函数 *****/
void InitLcd() {
    RS=0;RW=0;           //写指令
    E=1; P0=0x20;Delay(3); //设置4位数据接口
    E=0;
    WriteCommand(0x28);   //显示方式
    WriteCommand(0x06);   //显示光标移动设置
    WriteCommand(0x0c);   //显示开及光标关设置
    WriteCommand(0x01);   //清屏
}

/***** 自定义汉字字符函数 *****/
void Hz() {
    uchar i;
    WriteCommand(0x40);
    for (i=0;i<24;i++){
        WriteData(Hzzimo[i]);
    }
}

/***** 主函数 *****/
void main(void) {
    InitLcd();
    Delay(15);
    Hz();
    ePutstr(4,0,10,examp2);           // 第1行从第4位开始显示2014年5月26日
    ePutstr(0,1,16,"Lcd1602 test ok!"); // 第2行从第0位开始显示英文字符
    while(1);
}

```

➡ 6.5 12864点阵图型LCD接口技术

点阵字符型LCD显示模块只能显示英文字符和简单的汉字,要想显示较为复杂的汉字或图形,就必须采用点阵图型LCD显示模块。本节介绍12864点阵图型LCD显示模块与单片机的接口技术。

6.5.1 12864点阵图型LCD显示模块

12864点阵图型LCD显示模块内部控制器采用KS0108或HD61202。其引脚排列如图6.24所示,引脚功能见表6.19。

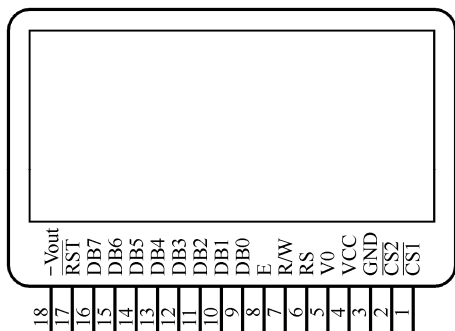


图6.24 12864点阵图型LCD显示模块引脚排列

表6.19 12864点阵图型LCD显示模块的引脚功能

引脚	符号	功能	引脚	符号	功能
1	$\overline{\text{CS1}}$	1: 选择左边64×64点	7	RW	1: 数据读取; 0: 数据写入
2	$\overline{\text{CS2}}$	1: 选择右边64×64点	8	E	使能信号, 负跳变有效
3	GND	地	9~16	DB0~DB7	数据信号
4	VCC	+5V电源	17	$\overline{\text{RST}}$	复位, 低电平有效
5	VO	显示驱动电源0~5V	18	-Vout	LCD驱动负电源
6	RS	1: 数据输入; 0: 命令输入	有些模块19、20脚为空脚		

12864点阵图型LCD内部存储器DDRAM与显示屏上的显示内容具有一一对应关系, 用户只要将显示内容写入到12864内部显示存储器DDRAM中, 就能实现正确显示。12864点阵图型LCD屏横向有128个点, 纵向有64个点, 分为左半屏和右半屏。DDRAM与显示屏的对应关系见表6.20。

表6.20 DDRAM与显示屏的对应关系

$\overline{\text{CS1}}=1$ (左半屏)						$\overline{\text{CS2}}=1$ (右半屏)					
Y=	0	1	...	62	63	0	1	...	62	63	行号
X=0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	0
	↓	↓	↓	↓	↓	↓	↓		↓	↓	↓
X=1	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	7
	↓	↓	↓	↓	↓	↓	↓		↓	↓	↓
X=7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	15
	↓	↓	↓	↓	↓	↓	↓		↓	↓	↓
...
X=7	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	DB0	56
	↓	↓	↓	↓	↓	↓	↓		↓	↓	↓
X=7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	DB7	63
	↓	↓	↓	↓	↓	↓	↓		↓	↓	↓

在12864点阵图型LCD屏上显示图形或汉字时, 可以利用字模提取软件获得图形或汉字的点阵代码。以“单”字16×16点阵显示为例, 按纵向取模方式获得的字模点阵代码为

```
DB 000H,000H,000H,0F0H,052H,054H,050H,0F0H
```

```
DB 050H,054H,052H,0F0H,000H,000H,000H,000H
DB 000H,008H,008H,00BH,00AH,00AH,00AH,07FH
DB 00AH,00AH,00AH,00BH,008H,008H,000H,000H
```

字模点阵数据是纵向的，一个像素对应一个位。8个像素对应一个字节。字节的位顺序是上低下高。例如，从上到下8个点的状态是“*-----”（*为黑点，-为白点），则转换的字模数据是41H（01000001B）。显示时，先输入汉字的上半部分16个数据，再输入下半部分16个数据。

12864点阵图型LCD显示模块的指令功能比较简单，共有8条指令。

① 读忙标志。编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	BUSY	0	ON/OFF	RESET	0	0	0	0

其中，BUSY=1，显示模块内部控制器忙，不能进行操作，只有BUSY=0时才允许操作。ON/OFF=1，显示关闭；ON/OFF=0，显示打开。RESET=1，复位状态；RESET=0，正常状态。在BUSY和RESET状态下，除读忙标志指令外，其他指令均不对液晶显示模块产生作用。

② 写指令。编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	0	下降沿	指令							

③ 写数据。编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
1	0	下降沿	显示数据							

操作时每完成一个列地址，计数器自动加1。

④ 显示开/关。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	1	1	1	1	D

其中，D=1，显示RAM中的内容；D=0，关闭显示。

⑤ 显示起始行。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	1	显示起始行（0~63）					

该指令规定显示屏上起始行对应DDRAM的行地址，有规律地改变显示起始行，可以实现显示滚屏的效果。

⑥ 页面地址。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	0	1	1	1	页面（0~7）		

DDRAM共64行，分8页，每页8行。

⑦ 列地址。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	显示列地址（0~63）					

列地址计数器在每一次读/写数据后自动加1，每次操作后明确起始列的地址。设置了页面地址和列地址，就唯一确定了DDRAM中的一个单元。这样单片机就可以用读/写指令读出该单元中的内容或向该单元写进一个字节数据。

⑧ 读数据。编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	1	显示数据							

该指令将DDRAM对应单元中的内容读出，然后列地址计数器自动加1。需要注意的是，进行读操作之前，必须有一次空读操作，紧接着再读才会读出所要求单元中的数据。

6.5.2 12864 LCD接口应用编程

单片机与12864图型LCD模块之间可以采用直接方式接口，也可以采用间接方式接口。

图6.25为采用间接方式实现8051单片机与12864图型LCD模块的接口电路。LCD模块的CS1、CS2、RS、R/W和E信号分别由8051单片机的P2.0、P2.1、P2.2、P2.3和P2.4来控制。由于间接控制方式需要通过单片机的端口引脚来操作液晶模块，因此在编写驱动程序时要特别注意时序的配合。例6-12列出了12864图型LCD模块间接方式接口的C51驱动程序。

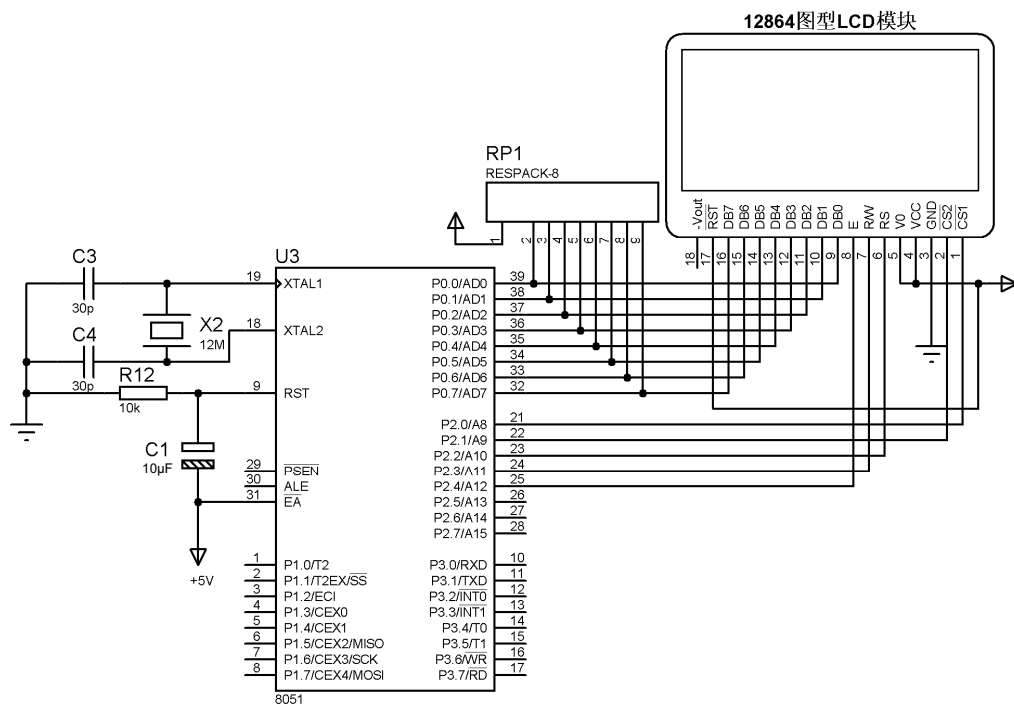


图6.25 采用间接方式实现8051单片机与12864图型LCD模块的接口电路

例6-12 12864图型LCD模块间接方式接口的C51驱动程序。

```
#include<reg51.h>
#include<absacc.h>
#include<intrins.h>
```

```

#define uchar unsigned char
#define uint unsigned int
#define PORT P0

sbit CS1=P2^0;
sbit CS2=P2^1;
sbit RS=P2^2;
sbit RW=P2^3;
sbit E=P2^4;
sbit bflag=P0^7;

uchar code Num[]={ //字模点阵数据
0x00,0x00,0x00,0xF0,0x52,0x54,0x50,0xF0, //单
0x50,0x54,0x52,0xF0,0x00,0x00,0x00,0x00,
0x00,0x08,0x08,0x0B,0x0A,0x0A,0x0A,0x7F,
0x0A,0x0A,0x0A,0x0B,0x08,0x08,0x00,0x00,
0x00,0x00,0x00,0x00,0xFC,0x20,0x20,0x20, //片
0x20,0x3E,0x20,0x20,0x20,0x30,0x20,0x00,
0x00,0x40,0x20,0x10,0x0F,0x01,0x01,0x01,
0x01,0x01,0x7F,0x00,0x00,0x00,0x00,0x00,
0x00,0x20,0x20,0xA0,0xFE,0xA0,0x20,0x00, //机
0xFC,0x04,0x04,0xFE,0x04,0x00,0x00,0x00,
0x00,0x04,0x02,0x01,0x7F,0x40,0x21,0x10,
0x0F,0x00,0x00,0x3F,0x40,0x40,0x78,0x00,
0x00,0x00,0x00,0xFC,0x04,0x04,0xE4,0xA4, //原
0xB4,0xAC,0xA4,0xA4,0xE4,0x06,0x04,0x00,
0x00,0x40,0x30,0x0F,0x20,0x10,0x0B,0x22,
0x42,0x3E,0x02,0x0A,0x13,0x30,0x00,0x00,
0x00,0x88,0x88,0xF8,0x88,0x88,0x00,0xFC, //理
0x24,0x24,0xFC,0x24,0x24,0xFE,0x04,0x00,
0x00,0x10,0x30,0x1F,0x08,0x48,0x40,0x4B,
0x49,0x49,0x7F,0x49,0x49,0x6B,0x40,0x00,
0x00,0x00,0x00,0xC0,0xBE,0x90,0x90,0x90, //与
0x90,0x90,0x90,0xD0,0x98,0x10,0x00,0x00,
0x00,0x04,0x04,0x04,0x04,0x04,0x04,0x04,
0x24,0x44,0x20,0x1F,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0xF8,0x08,0x88,0x08,0x2A, //应
0x4C,0x88,0x08,0x08,0x08,0xCC,0x08,0x00,
0x00,0x40,0x30,0x0F,0x20,0x20,0x23,0x2C,
0x20,0x23,0x30,0x2C,0x23,0x30,0x20,0x00,
0x00,0x00,0x00,0xFC,0x24,0x24,0x24,0x24, //用
0xFC,0x24,0x24,0x24,0xFE,0x04,0x00,0x00,
0x00,0x40,0x30,0x0F,0x02,0x02,0x02,0x02,
0x7F,0x02,0x22,0x42,0x3F,0x00,0x00,0x00
};

```

```
//清左半屏
```



```
void Left() {
    CS1=0; CS2=1;
}
//清右半屏
void Right() {
    CS1=1; CS2=0;
}
//判忙
void Busy_12864() {
    do{
        E=0; RS=0; RW=1;
        PORT=0xff;
        E=1; E=0; }while(bflag);
}
//命令写入
void Wreg(uchar c) {
    Busy_12864();
    RS=0; RW=0;
    PORT=c;
    E=1; E=0;
}
//数据写入
void Wdata(uchar c) {
    Busy_12864();
    RS=1; RW=0;
    PORT=c;
    E=1; E=0;
}
//设置显示初始页
void Pagefirst(uchar c) {
    uchar i=c;
    c=i|0xb8;
    Busy_12864();
    Wreg(c);
}
//设置显示初始列
void Linefirst(uchar c) {
    uchar i=c;
    c=i|0x40;
    Busy_12864();
    Wreg(c);
}
//清屏
void Ready_12864() {
    uint i,j;
    Left();
    Wreg(0x3f);
```

```

    Right();
    Wreg(0x3f);
    Left();
    for(i=0;i<8;i++){
        Pagefirst(i);
        Linefirst(0x00);
        for(j=0;j<64;j++){
            Wdata(0x00);
        }
    }
    Right();
    for(i=0;i<8;i++){
        Pagefirst(i);
        Linefirst(0x00);
        for(j=0;j<64;j++){
            Wdata(0x00);
        }
    }
}
//16×16汉字显示，纵向取模，字节倒序
void Display(uchar *s,uchar page,uchar line){
    uchar i,j;
    Pagefirst(page);
    Linefirst(line);
    for(i=0;i<16;i++){
        Wdata(*s); s++;
    }
    Pagefirst(page+1);
    Linefirst(line);
    for(j=0;j<16;j++){
        Wdata(*s); s++;
    }
}
//主函数
main(){
    Ready_12864();
    Left();
    Display(Num,0x03,0);
    Display(Num+32,0x03,16);
    Display(Num+64,0x03,32);
    Display(Num+96,0x03,48);
    Right();
    Display(Num+128,0x03,64);
    Display(Num+160,0x03,80);
    Display(Num+192,0x03,96);
    Display(Num+224,0x03,112);
    while(1);
}

```

6.6 T6963点阵图型LCD接口技术

内置T6963控制器的点阵图型LCD是较为常用且品种较多的一类点阵图形液晶显示模块。T6963是日本东芝公司的产品。它的最大特点是具有硬件初始值设置功能。其初始化工工作在加电时就已经基本完成，设计者开发软件的主要精力可以全部用在显示画面的设计上。

6.6.1 T6963点阵图型LCD显示模块

T6963 LCD显示模块内置有128种5×8点阵的ASCII字符发生器CGROM，并允许在显示存储器内开辟一个用户自定义的8×8点阵字模库CGRAM。T6963可以管理64KB的显示存储器，可以把显示存储器分成文本显示区、图形显示区及自定义字符库区等。

图6.26为内置T6963点阵图形LCD模块引脚排列，引脚功能见表6.21。

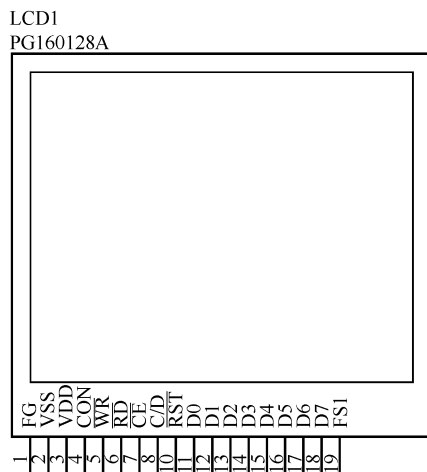


图6.26 内置T6963点阵图形LCD模块引脚排列

表6.21 内置T6963图形LCD模块的引脚功能

引 脚	功 能
FG	显示屏框架外壳地接地
Vss	电源地
VDD	+5V电源
CON	对比度调节负电压输入
\overline{WR}	数据写入，低电平有效
RD	数据读出，低电平有效
\overline{CE}	片选端，低电平有效
C/ \overline{D}	通道选择端，C/ \overline{D} =1为命令通道，C/ \overline{D} =0为数据通道
\overline{RST}	复位信号， \overline{RST} =1为正常工作， \overline{RST} =0为初始化T6963，文本和图形的地址、文本和图形区域设定被保持
D0~D7	数据总线
FS1	字体选择：FS = 1，选择6×8点阵字体；FS = 0，选择8×8点阵的字体

T6963提供两种命令形式：带参数命令和无参数命令。带参数命令中的参数需要在命令编码之前输入，格式为

参数1 参数2 命令编码

无参数命令只要给出命令编码即可。表6.22列出了T6963的全部命令编码。

表6.22 T6963的全部命令编码

命 令	参数1	参数2	编 码	功 能
寄存器设置	水平位置 (低7位有效)	垂直位置 (低5位有效)	0010 0001	设置光标位置
	偏置地址 (低5位有效)	00H	0010 0010	设置CGRAM偏置地址
	地址低8位	地址高8位	0010 0100	设置显示地址
显示区域设置	地址低8位	地址高8位	0100 0000	设置文本起始地
	列	00H	0100 0001	设置文本区宽度
	地址低8位	地址高8位	0100 0010	设置图形起始地址
	列	00H	0100 0011	设置图形区宽度
模式设定	—	—	1000 x000	文本与图形以“或”关系合成显示
	—	—	1000 x001	文本与图形以“异或”关系合成显示
	—	—	1000 x010	文本与图形以“与”关系合成显示
	—	—	1000 x011	文本显示特征以双字节表示
	—	—	1000 0xxx	内部CGROM模式
	—	—	1000 1xxx	外部CGRAM模式
显示模式	—	—	1001 0000	显示关闭
	—	—	1001 xx10	打开光标，黑色关闭
	—	—	1001 xx11	打开光标，黑色显示
	—	—	1001 01xx	打开文本方式，关闭图形方式
	—	—	1001 10xx	关闭文本方式，打开图形方式
	—	—	1001 11xx	图形文本混合方式
光标形式	—	—	1010 0000	1 条线
	—	—	1010 0001	2 条线
	—	—	1010 0010	3 条线
	—	—	1010 0011	4 条线
	—	—	1010 0100	5 条线
	—	—	1010 0101	6 条线
	—	—	1010 0110	7 条线
	—	—	1010 0111	8 条线
数据自动读写	—	—	1011 0000	数据自动写入设定
	—	—	1011 0001	数据自动读出设定
	—	—	1011 0000	自动复位
数据读写	—	—	1100 0000	数据写入，地址自动增量
	—	—	1100 0001	数据读出，地址自动增量

(续表)

命 令	参数1	参数2	编 码	功 能
数据读写	—	—	1100 0010	数据写入，地址自动减量
	—	—	1100 0011	数据读出，地址自动减量
	—	—	1100 0100	数据写入，地址保持不变
	—	—	1100 0101	数据读出，地址保持不变
屏幕读取	—	—	1110 0000	读取屏幕显示数据
屏幕拷贝	—	—	1110 1000	复制屏幕显示数据
位操作	—	—	1111 0xxx	位清零
	—	—	1111 1xxx	位置1
	—	—	1111 x000	位0
	—	—	1111 x001	位1
	—	—	1111 x010	位2
	—	—	1111 x011	位3
	—	—	1111 x100	位4
	—	—	1111 x101	位5
	—	—	1111 x110	位6
	—	—	1111 x111	位7

注：表中参数栏的“—”表示无参数。

T6963提供一个状态字，格式为

S7	S6	S5	S4	S3	S2	S1	S0
----	----	----	----	----	----	----	----

各位的含义见表6.23。

表6.23 T6963状态字的含义

状 态 位	含 义
S0	命令读/写状态，S0=1为准备好，S0=0为忙
S1	数据读/写状态，S1=1为准备好，S1=0为忙
S2	数据自动读状态，S2=1为准备好，S2=0为忙
S3	数据自动写状态，S3=1为准备好，S3=0为忙
S4	未用
S5	控制器运行检测可能性，S5=1为可能，S5=0为不能
S6	屏幕读取/屏幕拷贝出错状态，S6=1为出错，S6=0为正确
S7	闪烁状态检测，S7=1为显示，S6=0为关闭

这些标识位各有各的应用场合，并非同时有效。CPU在写命令一次读/写数据时，S0和S1要同时有效，即“准备好”状态。当CPU采用自动读/写功能时，S2或S3将取代S0和S1作为忙标志。S6是考察T6963屏幕读取和屏幕拷贝命令执行情况的标志位。S5和S7表示控制器内部的运行状态。T6963应用时不会使用它们。对T6963进行每一次软件操作之前都要判读忙标志，只有在在不忙（即“准备好”）状态下，CPU对T6963的操作才有效。T6963的读/写时序如图6.27所示。

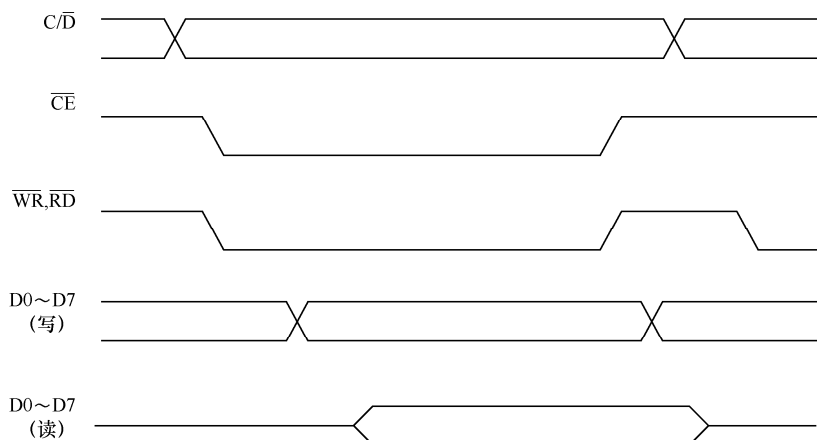


图6.27 T6963的读/写时序

6.6.2 T6963 LCD接口应用编程

图6.28为T6963点阵图形LCD模块与单片机的接口电路。LCD模块的 $\overline{C/D}$ 、 \overline{CE} 、 \overline{RD} 和 \overline{WR} 信号分别由8051单片机的P3.3、P3.5、P3.6和P3.7来间接控制，由于需要通过单片机的端口引脚来操作LCD显示模块，因此在编写驱动程序时要特别注意时序的配合。例6-13列出了T6963图型LCD模块的C51驱动程序。

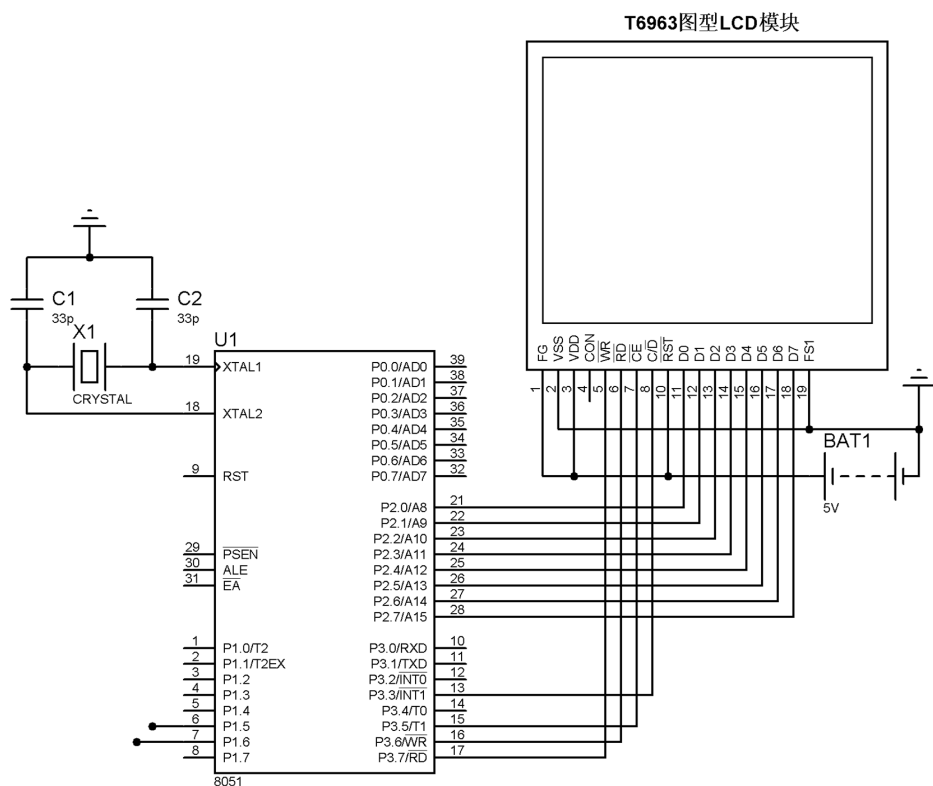


图6.28 T6963点阵图形LCD模块与单片机的接口电路

例6-13 T6963图型LCD模块的C51驱动程序。

```

#include<reg51.h>
#include<tg.h>
#define uchar unsigned char
#define uint unsigned int
#define High 1
#define Low 0

uchar data_display[12]={0};
uchar Etable[]={0x37,0x45,0x4c,0x43,0x4f,0x4d,0x45,0x00,0x35,
0x53,0x45,0x34,0x16,0x19,0x16,0x13,0x23};
Uchar code exprt1[11]=
    {0x80,0x82,0x00,0x84,0x86,0x00,0x88,0x8a,0x00,0x8c,0x8e};
Uchar code exprt3[11]=
    {0x81,0x83,0x00,0x85,0x87,0x00,0x89,0x8b,0x00,0x8d,0x8f};
uchar code exprt2[18]={0x90,0x92,0x94,0x96,0x98,0x9a,0x9c,
0x9e,0xa0,0xa2,0xa4,0xa6,0xa8,0xaa,0xac,0xae,0xb0,0xb2};
uchar code exprt4[18]={0x91,0x93,0x95,0x97,0x99,0x9b,0x9d,
0x9f,0xa1,0xa3,0xa5,0xa7,0xa9,0xab,0xad,0xaf,0xb1,0xb3};
uchar code exprt5[12]=
    {0xb4,0xb6,0xbc,0xbe,0xc0,0xc2,0xc4,0xc6,0xc8,0xca,0xcc,0xce};
uchar code exprt7[12]=
    {0xb5,0xb7,0xbd,0xbf,0xc1,0xc3,0xc5,0xc7,0xc9,0xcb,0xcd,0xcf};
uchar code exprt6[12]=
    {0xb8,0xba,0xbc,0xbe,0xc0,0xc2,0xc4,0xc6,0xc8,0xca,0xcc,0xce};
uchar code exprt8[12]=
    {0xb9,0xbb,0xbd,0xbf,0xc1,0xc3,0xc5,0xc7,0xc9,0xcb,0xcd,0xcf};
sbit cd=P3^3;
sbit ce=P3^5;
sbit rd=P3^6;
sbit wr=P3^7;
sbit cs=P1^7;
sbit sclk=P1^6;
sbit dout=P1^5;
sbit cs1=P1^3;

/***** 延时函数 *****/
void Delay_nms(uchar n){
    uchar a;
    for(;n>0;n--){
        for(a=0;a<200;a++) {
            ;
        };
    };
}

/***** 检查s0s1函数 *****/

```

```

void Check_s0s1(void) {
    uchar temp=0;
    ce=Low;
    while(1) {
        P2=0xff;
        cd=High;
        rd=Low;
        wr=High;
        rd=High;
        temp=P2;
        if((temp&0x03)==0x03) break;
    };
    ce=High;
}

/***** 检查s3函数 *****/
void Check_s3(void) {
    uchar temp=0;
    ce=Low;
    while(1) {
        P2=0xff;
        cd=High;
        rd=Low;
        wr=High;
        rd=High;
        temp=P2;
        if((temp&0x04)==0x04) break;
    };
    ce=High;
}

/***** 命令写入函数 *****/
void Write_command(uchar command) {
    cd=High;
    ce=Low;
    wr=Low;
    P2=command;
    wr=High;
}

/***** 数据写入函数 *****/
void Write_data(uchar udata) {
    cd=Low;
    ce=Low;
    wr=Low;
    P2=udata;
    wr=High;
}

```



```

}

/***** 自动数据写入函数 *****/
void Auto_writedata(uchar *auto_pointer,uint length) {
    uint a;
    Check_s0s1(); //状态检测
    Write_command(0xb0); //连续写入开始
    for(a=0;a<length;a++) {
        Check_s3(); //状态检测
        Write_data(*(auto_pointer+a));
    };
    Check_s0s1(); //状态检测
    Write_command(0xb2); //连续写入结束
}

/***** 单参数命令函数 *****/
void Single_parameter_cmd(uchar sdata,uchar command2) {
    Check_s0s1();
    Write_data(sdata);
    Check_s0s1();
    Write_command(command2);
}

/***** 双参数命令函数 *****/
void Two_parameter_cmd(uchar tdata1,uchar tdata2,uchar command3) {
    Check_s0s1();
    Write_data(tdata1);
    Check_s0s1();
    Write_data(tdata2);
    Check_s0s1();
    Write_command(command3);
}

/***** 清除RAM函数 *****/
void Clear_RAM(void) {
    uint cloop;
    Two_parameter_cmd(0x00,0x00,0x24);
    for(cloop=0;cloop<0x3000;cloop++)
        Single_parameter_cmd(0,0xc0);
}

/***** T6963初始化函数 *****/
void T6963_initialize(void) {
    Clear_RAM();
    Two_parameter_cmd(0x00,0x00,0x40); //设置文本起始位置
    Two_parameter_cmd(0x14,0x00,0x41); //设置宽度
    Two_parameter_cmd(0x00,0x08,0x42); //设置图形起始位置
}

```

```

        Two_parameter_cmd(0x14,0x00,0x43);           //设置宽度
        Check_s0s1();
        Write_command(0x80);                         //使用内部ROM
        Check_s0s1();
        Write_command(0xa0);                         //光标大小
        Check_s0s1();
        Write_command(0x94);                         //开启文本显示
    }

    /***** 设置CGRAM函数 *****/
    void Set_CGRAM(void){
        Two_parameter_cmd(0x05,0x00,0x22);           //设置偏置寄存器
        Two_parameter_cmd(0x00,0x2c,0x24);           //设置访问地址
        Auto_writedata(Ctable,672);                 //设置CGRAM字符内容
    }

    /***** 文本显示函数 *****/
    void Textdisplay(void) {
        uchar i=0;
        // 英文显示
        Two_parameter_cmd(45,0x00,0x24);             //显示位置
        for(i=0;i<11;i++) {
            Single_parameter_cmd(Etable[i],0xc0);     //显示内容
            Delay_nms(40);
        }
        Two_parameter_cmd(87,0x00,0x24);
        for(i=11;i<16;i++) {
            Single_parameter_cmd(Etable[i],0xc0);
            Delay_nms(40);
        }
        // 中文显示
        Two_parameter_cmd(145,0x00,0x24);
        for(i=0;i<11;i++) {
            Single_parameter_cmd(exprt1[i],0xc0);
            Delay_nms(10);
        }
        Two_parameter_cmd(165,0x00,0x24);
        for(i=0;i<11;i++) {
            Single_parameter_cmd(exprt3[i],0xc0);
            Delay_nms(10);
        }
        Two_parameter_cmd(201,0x00,0x24);
        for(i=0;i<18;i++) {
            Single_parameter_cmd(exprt2[i],0xc0);
            Delay_nms(10);
        }
        Two_parameter_cmd(221,0x00,0x24);
    }

```

```

        for(i=0;i<18;i++) {
            Single_parameter_cmd(exprt4[i],0xc0);
            Delay_nms(10);
        }
        Two_parameter_cmd(89,0x01,0x24);
        for(i=0;i<11;i++)
            Single_parameter_cmd(Etable[i],0xc0);
        Two_parameter_cmd(131,0x01,0x24);
        for(i=11;i<16;i++)
            Single_parameter_cmd(Etable[i],0xc0);
// 中文显示
        Two_parameter_cmd(189,0x01,0x24);
        for(i=0;i<11;i++)
            Single_parameter_cmd(exprt1[i],0xc0);
        Two_parameter_cmd(209,0x01,0x24);
        for(i=0;i<11;i++)
            Single_parameter_cmd(exprt3[i],0xc0);
        Two_parameter_cmd(245,0x01,0x24);
        for(i=0;i<18;i++)
            Single_parameter_cmd(exprt2[i],0xc0);
        Two_parameter_cmd(0x09,0x02,0x24);
        for(i=0;i<18;i++)
            Single_parameter_cmd(exprt4[i],0xc0);
    }

/***** 图形显示函数 *****/
void Graphic_display(void) {
    Two_parameter_cmd(0xa0,0x08,0x24);           //图形显示位置
    Auto_writedata(Graphic1,2400);               //图形显示内容
    Two_parameter_cmd(0xa0,0x12,0x24);
    Auto_writedata(Graphic1,2400);
    Two_parameter_cmd(0xc0,0x1a,0x24);           //向上滚动显示
    Auto_writedata(Graphic1,2400);
}

/***** 文本右滚显示函数 *****/
void Textscroll_right(void) {
    uchar rloop=0;
    for(rloop=20;rloop>0;rloop--) {
        Two_parameter_cmd(rloop,0x00,0x40);
        Delay_nms(250);
    }
}

/***** 文本上滚显示函数 *****/
void Textscroll_up(void) {
    uchar uloop=0;

```

```

        uint up_address;
        for(uloop=0;uloop<15;uloop++) {
            up_address=uloop*20;
            Two_parameter_cmd(up_address&0x00ff, (up_address&0xff00)>>8,0x40);
            Delay_nms(250);
        }
    }

    /***** 图形左滚显示函数 *****/
    void Graphic_scroll_left(void) {
        uint Glloop;
        for (Glloop=0;Glloop<40;Glloop++){ //滚动的页数
            Two_parameter_cmd(Glloop,0x08,0x42);
            Delay_nms(250);
        }
    }

    /***** 主函数 *****/
    main() {
        cs=High;
        cs1=High;
        T6963_initialize();
        Set_CGRAM();
        Textdisplay();
        Textscroll_up();
        Textscroll_right();
        Write_command(0x98);
        Graphic_display();
        Graphic_scroll_left();
        while(1);
    }

```

模数与数模转换接口应用

7.1 转换器的主要技术指标

模/数转换器ADC的主要技术指标如下。

① 分辨率 (Resolution)。分辨率反映转换器所能分辨的被测量最小值。通常用输出二进制代码的位数来表示。例如, 分辨率为8位的ADC, 模拟电压的变化范围被分成 2^8-1 级 (255级), 而分辨率为10位的ADC, 模拟电压的变化范围被分成 $2^{10}-1$ 级 (1023级)。因此, 同样范围的模拟电压, 用10位A/D转换器所能测量的被测量最小值要比用8位ADC小得多。

② 精度 (precision)。精度是指转换结果相对于实际值的偏差, 有绝对精度和相对精度两种表示方法。绝对精度用二进制代码的最低位 (LSB) 的倍数来表示, 如 $\pm (1/2) \text{ LSB}$ 、 $\pm 1 \text{ LSB}$ 等。相对精度用绝对精度除以满量程值的百分数来表示, 如 $\pm 0.05\%$ 等。

应当指出, 分辨率和精度是两个不同的概念。同样分辨率ADC的精度可能不同。例如, ADC0804与AD570, 分辨率均为8位, 但ADC0804的精度为 $\pm 1 \text{ LSB}$, 而AD570的精度为 $\pm 2 \text{ LSB}$ 。因此, 分辨率高, 但精度不一定高; 而精度高, 则分辨率必然也高。

③ 量程 (满刻度范围——FULL Scale Range)。量程是指输入模拟电压的变化范围。例如, 某转换器具有10V的单极性范围或 $-5 \sim +5 \text{ V}$ 的双极性范围, 则它们的量程都为10V。应当指出, 满刻度只是个名义值, 实际上转换器的最大输出值总是比满刻度值小 $1/2^n$, n 为转换器的位数。这是因为模拟量的0值是 2^n 个转换状态中的一个, 在0值以上只有 2^n-1 个梯级。但按通常习惯, 转换器的模拟量范围总是用满刻度表示。例如, 12位的ADC, 其满刻度值为10V, 而实际的最大输出值为

$$10 - 10 \times \frac{1}{2^{12}} = 10 \times \frac{4095}{4096} = 9.9976(\text{V})$$

④ 线性度误差 (Linearity Error)。理想的转换器特性应该是线性的, 即模拟量输入与数字量输出成线性关系。线性度误差是指转换器实际的模拟—数字转换关系与理想的直线关系不同而出现的误差, 通常用多少LSB表示。

⑤ 转换时间 (Conversion Time)。从发出启动转换开始直至获得稳定的二进制代码所需的时间被称为转换时间。转换时间与转换器工作原理及其位数有关。同种工作原理的转换

器，通常位数越多，其转换时间越长。

数/模转换器DAC的主要技术指标与模/数转换器ADC基本相同，只是转换时间的概念略有不同。DAC的转换时间又叫建立时间，是指当输入的二进制代码从最小值突然跳变至最大值时，模拟输出电压相应的满度跳跃并达到稳定所需的时间。一般而言，DAC的转换时间比ADC要短得多。

7.2 数/模转换器DAC接口技术

数/模转换器DAC (Digital-Analog Converter) 的功能是将数字量转换为与其成比例的模拟电压或电流信号，输出到仪表外部进行各种控制。本节主要介绍DAC芯片的使用方法及其与单片机的接口技术。DAC芯片种类繁多，有通用廉价的DAC芯片，也有高速、高精度及高分辨率的DAC芯片。表7.1列出了几种常用DAC芯片的特点及性能。

表7.1 几种常用DAC芯片的特点及性能

	位数	建立时间 (转换时间) (ns)	非线性误差 (%)	工作电压 (V)	基准电压 (V)	功耗 (mV)	与TTL兼容
DAC0832	8	1000	0.2~0.05	+5~+15	-10~+10	20	是
AD7524	8	500	0.1	+5~+15	-10~+10	20	是
AD7520	10	500	0.2~0.05	+5~+15	-25~+25	20	是
AD561	10	250	0.05~0.025	V _{cc} +5~+16 V _{ee} -10~-16	/	正电源8~10 负电源12~14	是
AD7521	12	500	0.2~0.05	+5~+15	-25~+25	20	是
DAC1210	12	1000	0.05	+5~+15	-10~+10	20	是

各种类型的DAC芯片都具有数字量输入端和模拟量输出端及基准电压端。数字输入端有以下几种类型：①无数据锁存器；②带单数据锁存器；③带双数据锁存器；④可接收串行数字输入。第①种在与单片机接口时，要外加锁存器，第②种和第③种可直接与单片机接口，第④种与单片机接口十分简单，接收数据较慢，适用于远距离现场控制的场合。模拟量输出有两种方式：电压输出和电流输出。电压输出的DAC芯片相当于一个电压源，内阻很小。选用这种芯片时，与它匹配的负载电阻应较大。电流输出的芯片相当于电流源，内阻较大。选用这种芯片时，负载电阻不可太大。

在实际应用中，常选用电流输出的DAC芯片实现电压输出，如图7.1所示。图中，(a)为反相输出，输出电压为 $V_{OUT} = -iR$ ，(b)为同相输出，输出电压为 $V_{OUT} = -iR \times \left(1 + \frac{R_2}{R_1}\right)$ 。

上述两种电路均是单极性输出，如0~+5V、0~+10V。在实际应用中，有时需要双极性输出，如±5V、±10V，这时可采用如图7.1(c)所示的电路。图中， $R_3=R_4=2R_2$ ，输出电压 V_{OUT} 与基准电压 V_{REF} 及第一级运放 A_1 输出电压 V_1 的关系是 $V_{OUT} = -(2V_1 + V_{REF})$ 。 V_{REF} 通常就是芯片的电源电压或基准电压，极性可正、可负。

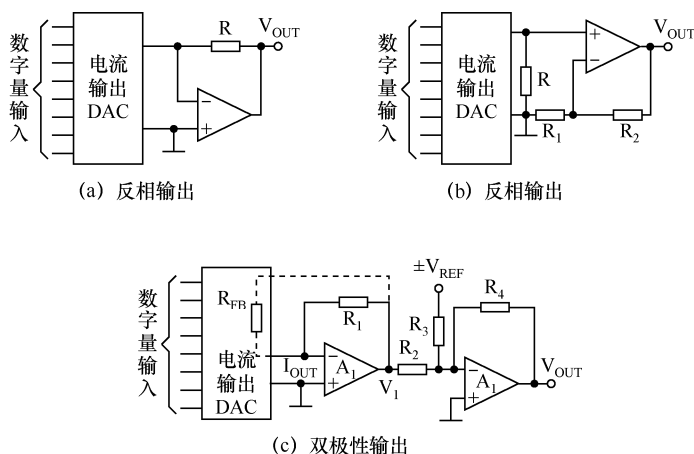


图7.1 将电流型DAC芯片连接成电压输出方式

7.2.1 DAC0832接口应用编程

DAC0832是典型的带内部双缓冲数据缓冲器的8位D/A芯片。其逻辑结构如图7.2所示。图中， \overline{LE} 是寄存命令，当 $\overline{LE}=1$ 时，寄存器输出随输入变化，当 $\overline{LE}=0$ 时，数据锁存在寄存器中，不再随数据总线上数据的变化而变化。当 \overline{ILE} 端为高电平， \overline{CS} 和 $\overline{WR1}$ 同时为低电平时， $\overline{LE1}=1$ ；当 $\overline{WR1}$ 变为高电平时，输入寄存器便将输入数据锁存。当 \overline{XFER} 和 $\overline{WR2}$ 同时为低电平时， $\overline{LE2}=1$ ，DAC寄存器的输出随寄存器的输入变化， $\overline{WR2}$ 上升沿将输入寄存器的信息锁存在该寄存器中。 R_{FB} 为外部运算放大器提供的反馈电阻。 V_{REF} 是由外电路为芯片提供一的 $+10V \sim -10V$ 的基准电源。 I_{OUT1} 和 I_{OUT2} 是电流输出端，两者之和为常数。

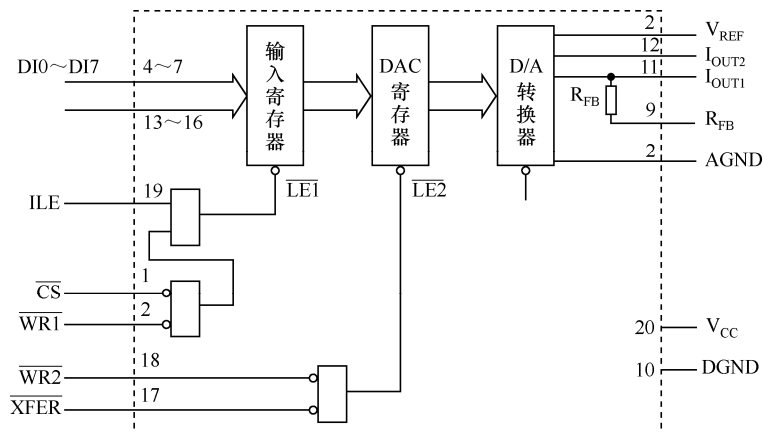


图7.2 DAC0832逻辑结构

图7.3为DAC0832与8051单片机组成的D/A转换接口Proteus仿真电路。其中，DAC0832工作在单缓冲器方式， \overline{ILE} 接 $+5V$ ， \overline{CS} 和 \overline{XFER} 相连后由8051的P2.7控制， $\overline{WR1}$ 和 $\overline{WR2}$ 相连后由8051的 \overline{WR} 控制。例7-1为采用C51编写的驱动程序，程序执行后DAC将产生输出电压驱动直流电机运转，通过“加速”和“减速”按键调节DAC输出不同电压，可使直流电机以不同速度运转。

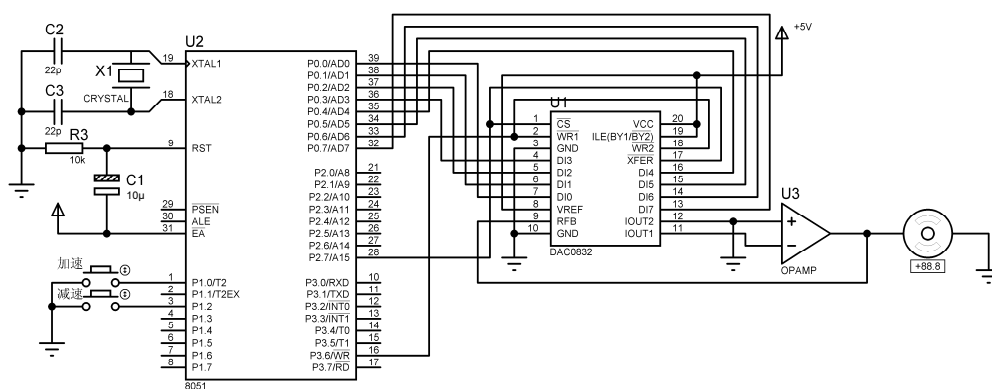


图7.3 DAC0832与8051单片机组成的D/A转换接口Proteus仿真电路

例7-1 采用DAC输出驱动直流电动机的C51程序。

```
#include<reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define uint unsigned int
#define DAC0832 0x7fff //定义DAC0832地址

sbit K1=P1^0;           //定义按键
sbit K2=P1^2;
uchar Dval=0x20;
uint i=0x8000;

/***** 加速处理函数 *****/
void INCDAC () {
    Dval=Dval+0x10;
    if(Dval>=0xe0)
        Dval=0xe0;
}

/***** 减速处理函数 *****/
void DECDAC () {
    Dval=Dval-0x10;
    if(Dval<=0x20)
        Dval=0x20;
}

/***** 主函数 *****/
main() {
    while(1) {
        XBYTE[DAC0832]=Dval;    //启动DAC0832
        if (K1==0) {             //判断加速键是否按下
            while(i--);          //延时反弹跳
            if (K1==0) INCDAC (); //加速
        }
    }
}
```



```

        i=0x8000;
    }
    if (K2==0) {                                //判断减速键是否按下
        while(i--);                            //延时反弹跳
        if (K2==0) DECDAC();                  //减速
        i=0x8000;
    }
}

```

图7.4为具有两路模拟量输出的DAC0832与8051单片机的接口。两片DAC0832工作在双缓冲器方式以实现两路同步输出。两片DAC0832的 \overline{CS} 分别连到8051的P2.0和 $\overline{P2.0}$ ，两片DAC0832的 \overline{XFER} 都连到P2.7，两片的 $\overline{WR1}$ 和 $\overline{WR2}$ 都连到 \overline{WR} ，这样两片DAC0832的数据输入锁存器分别被编址为0FEFFH和0FFFFH，而它们的DAC寄存器地址都是7FFFH。例7-2是采用C51编写的驱动程序，执行后，可以同时使两路DAC产生不同输出电压驱动直流电动机，也可以利用这两路模拟量输出分别控制CRT显示器的x、y偏转，实现特殊要求的显示。

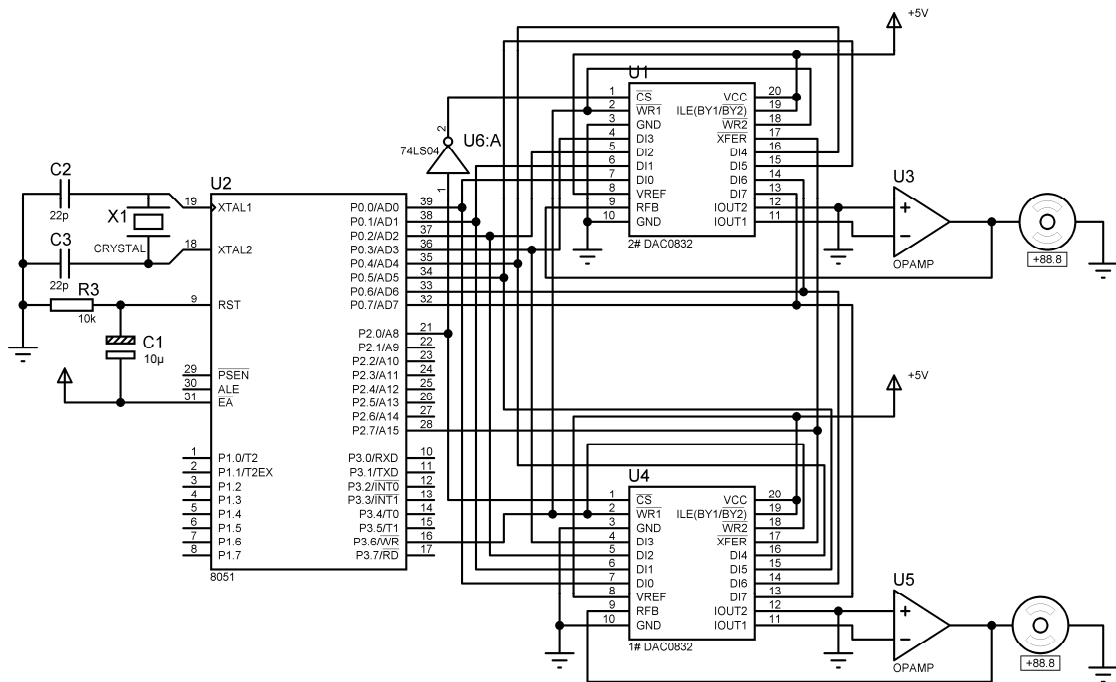


图7.4 具有两路模拟量输出的DAC0832与8051单片机的接口

例7-2 两路DAC同步输出的C51驱动程序。

```
#include <reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define DAC1 0xfeff //定义DAC1的数据地址
#define DAC2 0xffff //定义DAC2的数据地址
#define DAC 0x7fff //定义DAC输出地址
```

```

uchar Dval1=0x20;
uchar Dval2=0xf0;
/***** 主函数 *****/
main() {
    XBYTE[DAC1]=Dval1;    //给DAC1送数据x
    XBYTE[DAC2]=Dval2;    //给DAC2送数据y
    XBYTE[DAC]=Dval2;     //同时启动DAC1和DAC2
    while(1);
}

```

如果要设计具有多路模拟量输出的DAC接口，则可以仿照图7.4的方法，采用多个DAC与单片机接口，也可以采用多路输出复用一個DAC芯片的设计方法。图7.5为4通道模拟量输出共享一个DAC0832芯片的接口电路。

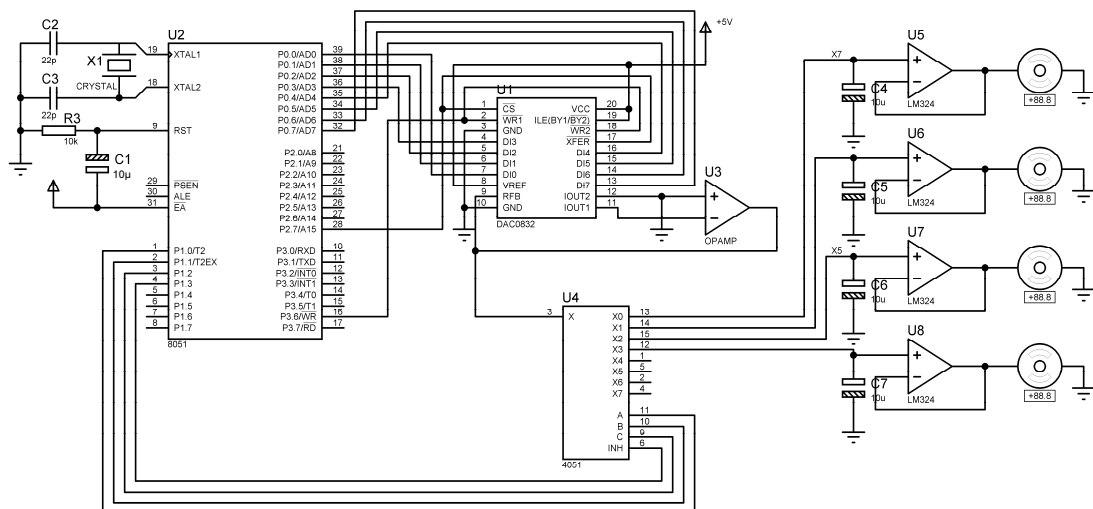


图7.5 4通道模拟量输出共享一个DAC0832芯片的接口电路

单片机送来的数字信号先经由DAC0832转换成模拟电压，再由多路开关4051分时加至保持运算放大器LM324的输入端，并将电压存储在电容器中。为了使保持器有稳定的输出信号，应对保持电容定时刷新，使电容上的电压始终与单片机输出的数据保持一致。刷新时，每一回路接通的时间取决于多路开关的断路电阻、运放的输入电阻、保持电容的容量等。由于保持电容上的输入电压不可避免地存在微量泄漏，因此这种接口电路的通道数不宜太多。将模拟电压输出数据存放在片内RAM单元中，例7-3是利用1片DAC输出4路不同电压的C51驱动程序，执行后可完成4个通道D/A转换，分别输出4路不同电压驱动4台电动机以不同速度旋转。

例7-3 利用1片DAC输出4路不同电压的C51驱动程序。

```

#include <reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define uint unsigned int

```

```

#define DAC 0x7fff //定义DAC输出地址

uchar Dval[]={0x50,0x80,0xc0,0xf0};

/***** 延时函数 *****/
void delay(){
    uint i;
    for(i=0;i<35000;i++);
}

/***** 主函数 *****/
main(){
    uchar *ptr,j,DP;
    while(1){
        ptr=Dval; DP=0x00;
        for(j=0;j<4;j++){ //4个通道
            P1=DP; //选通多路开关
            XBYTE[DAC]=*ptr; //启动DAC
            delay();
            ptr++;DP++;
        }
    }
}

```

7.2.2 DAC1208接口应用编程

DAC0832是8位分辨率的D/A芯片，与8位单片机接口容易，但有时会显得分辨率不够。下面介绍一种带内部锁存器的12位分辨率DAC芯片DAC1208。图7.6为DAC1208的逻辑结构框图。

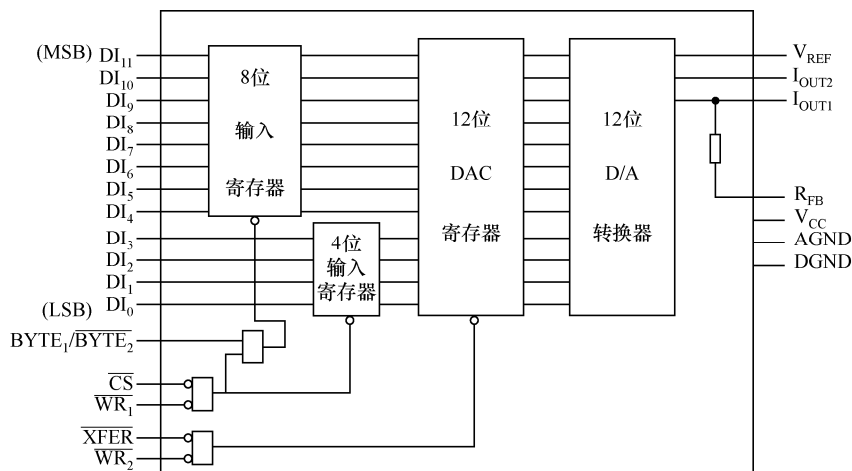


图7.6 DAC1208的逻辑结构框图

与DAC0832相似，DAC1208也是双缓冲器结构，输入控制线与DAC0832也很相似， \overline{CS} 和 \overline{WR}_1 用来控制输入寄存器， \overline{XFER} 和 \overline{WR}_2 用来控制DAC寄存器，但增加了一条控制线 $\overline{BYTE1}/\overline{BYTE}_2$ ，用来区分输入8位寄存器和4位寄存器，当 $\overline{BYTE1}/\overline{BYTE}_2=1$ 时，两个寄存器都被选中， $\overline{BYTE1}/\overline{BYTE}_2=0$ 时，只选中4位输入寄存器。DAC1208与8051单片机的接口如图7.7所示。DAC1208的 \overline{CS} 端接8051的P2.7，DAC1208的 $\overline{BYTE1}/\overline{BYTE}_2$ 端接8051的P2.0，因此DAC1208的8位输入寄存器地址为7FFFH，4位输入寄存器地址为7EFFH；8051的P2.7反向后接DAC1208的 \overline{XFER} 端，因此DAC1208的DAC寄存器地址为FFFFH。DAC1208采用双缓冲器工作方式，送数时，应先送高8位数据 $DI_{11}\sim DI_4$ ，再送低4位数据 $DI_3\sim DI_0$ ，送完12位数据后，再打开DAC寄存器，设12位数据存放在内部RAM区的40H和41H单元中，高8位存于40H，低4位存于41H。下面的例7-4是采用C51编写的驱动程序，执行后，可完成一次12位D/A转换，并利用DAC1208输出电压驱动直流电动机。

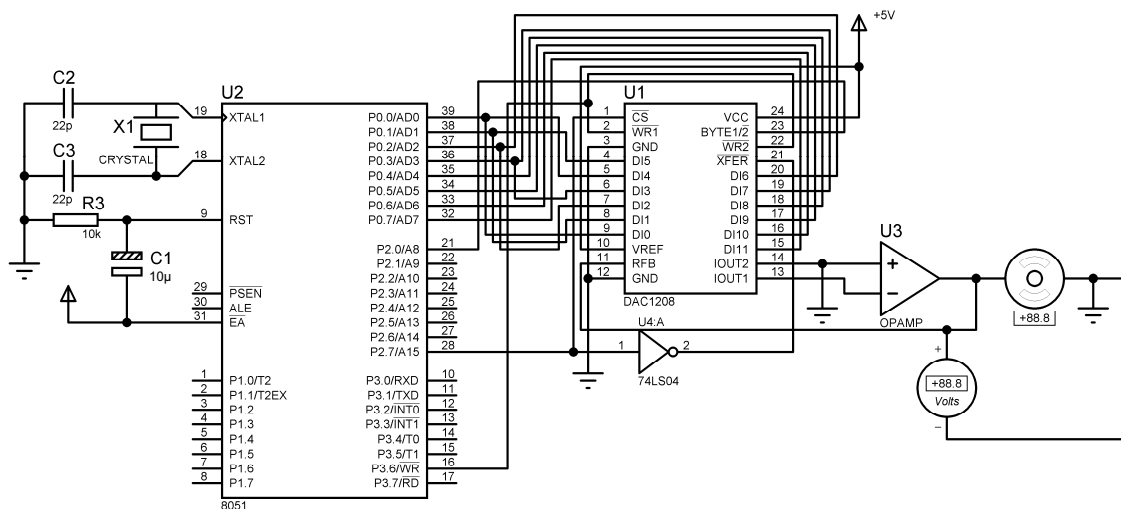


图7.7 DAC1208与8051单片机的接口

例7-4 DAC1208的C51驱动程序。

```
#include <reg52.h>
#include <absacc.h>

#define DAC8 0x7fff //定义1208高8位输入寄存器地址
#define DAC4 0x7eff //定义1208低4位输入寄存器地址
#define DAC 0xffff //定义1208DAC寄存器地址

/***** 主函数 *****/
main() {
    XBYTE[DAC8]=0xff; //输出高8位数据
    XBYTE[DAC4]=0x0f; //输出低8位数据
    XBYTE[DAC]=0x0f; //启动12位D/A转换
    while(1);
}
```

7.2.3 串行D/A芯片TLC5615接口应用编程

并行DAC转换时间短，反应速度快，但芯片引脚多，体积较大，与单片机的接口电路较复杂。因此在一些对DAC转换时间没有太高要求的场合，可以选用串行DAC芯片，其转换时间虽然比并行DAC稍长，但芯片引脚少，与单片机的接口电路简单，体积小，价格低。美国TI公司推出的TLC5615是一种串行10位DAC芯片，只需要3根串行总线就可以完成10位数据的输入，易于和单片机接口。TLC5615采用单5V电源工作，高阻抗基准输入端，上电时内部自动复位，最大功耗为1.75mW，转换速率快，更新率为1.21MHz。

图7.8为TLC5615的内部功能框图，主要组成部分包括：10位DAC电路；一个16位移位寄存器（分为高4位虚拟位、10位数据位、低2位填充位），接收串行二进制数，并且有一个级联的数据输出端DOUT；并行输入、输出的10位DAC寄存器，为10位DAC电路提供待转换的二进制数据；电压跟随器为参考电压端REFIN提供很高的输入阻抗，大约为10M Ω ； $\times 2$ 电路提供最大值为2倍于REFIN的输出；上电复位电路和控制电路。

图7.9列出了TLC5615的引脚分布，各引脚功能见表7.2。

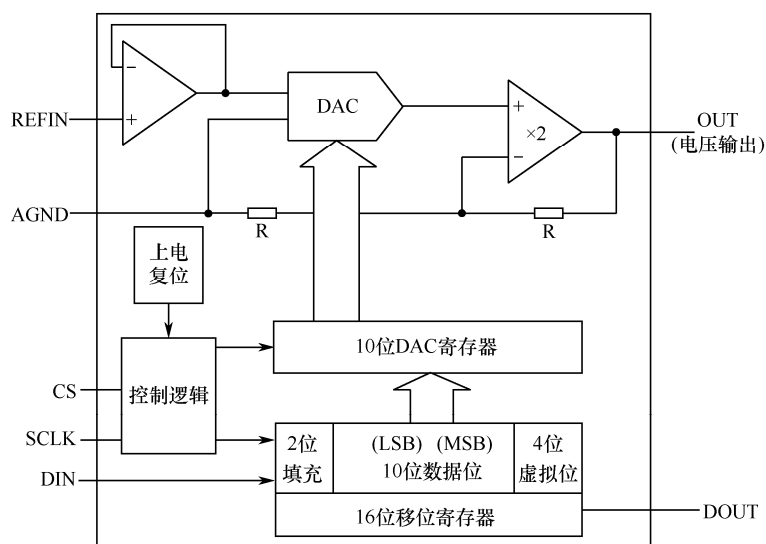


图7.8 TLC5615的内部功能框图

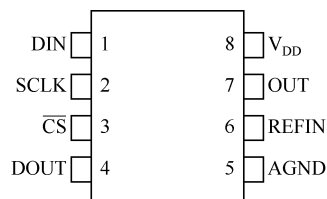


图7.9 TLC5615的引脚分布

表7.2 TLC5615的引脚功能

引 脚	功 能
DIN	串行数据输入端
SCLK	串行时钟输入端
\overline{CS}	片选端，低电平有效
DOUT	用于级联时的串行数据输出端
V _{DD}	正电源端，通常取+5V
OUT	DAC模拟电压输出端
REFIN	基准电压输入端2V~(V _{DD} -2V)
AGND	模拟地

TLC5615具有12位数据序列和16位数据序列两种工作方式。单片TLC5615工作时采用12位数据序列，在 \overline{CS} 为低电平期间，由时钟信号SCLK控制串行数据DIN向16位移位寄存器依次输入10位有效数据位和低2位填充位（填充位数据任意），高位在前，低位在后，需要12个SCLK时钟完成一次数据传输。

多片TLC5615以级联方式（本片的DOUT接到下一片的DIN）工作时采用16位数据序列，在 \overline{CS} 为低电平期间，由时钟信号SCLK控制串行数据DIN向16位移位寄存器依次输入高4位虚拟位、10位有效数据位和低2位填充位（填充位数据任意），高位在前，低位在后，由于增加了高4位虚拟位，所以需要16个SCLK时钟完成一次数据传输。

无论采用12位数据序列还是16位数据序列工作方式，输出电压均为

$$V_{OUT} = V_{REFIN} \times N / 1024$$

其中， V_{REFIN} 是参考电压， N 为输入的二进制数。

TLC5615采用SPI方式传输数据，最大传输速度为1.21MHz，D/A转换时间为12.5 μ s，故一次写入数据（ \overline{CS} 引脚从低电平至高电平跳跃）后，必须延时15 μ s左右才可第二次写入数据再次启动D/A转换。

TLC5615与单片机8051的接口电路如图7.10所示，例7-5为C51驱动程序。本例采用12位数据序列，数字量与输出模拟电压呈线性，0x0000对应输出电压为0V；0x0ffc对应输出电压为 $2 \times V_{REFIN}$ ， V_{REFIN} 的范围为0~($V_{DD}-2$)。程序执行后，可从OUT端输出锯齿波电压，连接到REFIN端电位器用于调节输出电压幅度。

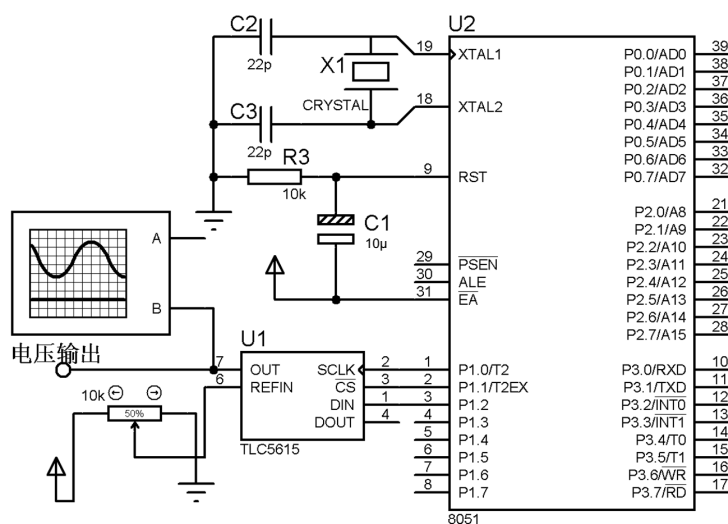


图7.10 TLC5615与单片机的接口电路

例7-5 TLC5615的C51驱动程序。

```
#include "reg51.h"
#define uint unsigned int
#define uchar unsigned char

/***** 引脚定义 *****/
sbit SCLK = P1^0;
```

```

sbit CS  = P1^1;
sbit DAT = P1^2;
uint k, DACdata; //定义需要转换的数据量, 改变它的值, 得到不同的模拟电压

/***** D/A转换函数 *****/
void DA_Conver(uint DAValue){
    uchar i;
    DAValue <= 4;
    CS = 0; //选中TLC5615芯片
    SCLK = 0;
    //12个时钟周期内, 前10个时钟为10位DA数据, 后两个时钟为填充字节
    for(i = 0; i < 12; i++) {
        DAT = (bit)(DAValue & 0x8000);
        SCLK = 1;
        DAValue <= 1;
        SCLK = 0;
    }
    CS = 1; //CS上升和下降沿只在SCLK为低时才有效
    SCLK = 0;
}

/***** 主函数 *****/
void main(){
    DACdata=0; //准备D/A转换数据
    while(1){
        DACdata=k<<2;
        k++;
        DA_Conver(DACdata); //启动D/A转换
        if(k==0x3ff) k=0;
    }
}

```

7.2.4 利用DAC接口实现波形发生器

利用DAC接口输出的模拟量（电压或电流）可以在许多场合得到应用。本节介绍DAC接口的一种应用——波形发生器，可以在8051单片机的控制下，产生三角波、锯齿波、方波及正弦波。各种波形所采用的硬件接口都是一样的，由于控制程序不同而产生不同的波形。采用7.2.1节中的图7.3所示硬件接口，DAC0832的地址为7FFFH，工作于单缓冲器方式，执行一次对DAC0832的写入操作即可完成一次D/A转换。8051单片机的累加器A从0开始循环增量，每增量一次向DAC0832写入一个数据，得到一个输出电压，这样可以获得一个正向的阶梯波，如图7.11所示。

DAC0832的分辨率为8位，如其满度电压为5V，则一个阶梯的幅度为

$$\Delta V = \frac{5V}{2^8} = \frac{5V}{256} = 19.5\text{mV}$$

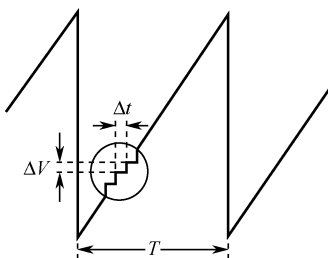


图7.11 正向阶梯波

由图7.11可见，由于每一个阶梯波较小，因此总体看来就是一个锯齿波。如果要改变波形的周期，则可采用软件延时的方法来实现，在延时子程序中改变延时时间的长短，即可改变输出波形的周期。用这种方法来产生波形，其频率是较低的。要想提高频率，可通过改进程序来减少执行时间，但效果有限，根本的办法还得靠改进硬件电路。

如果想获得任意起始电压和终止电压的波形，则需要确定起始电压和终止电压所对应的数字量。程序中，从起始电压对应的数字量开始输出，当达到终止电压对应的数字量时返回，如此反复。

如果将正向锯齿波与负向锯齿波组合起来就可以获得三角波。

方波信号也是波形发生器中常用的一种信号，通过调整延时时间可得到不同占空比的矩形波。图7.12为占空比为 $T_1/(T_1+T_2)$ 的矩形波，改变延时值使 $T_1=T_2$ 即可得到方波。

利用DAC接口实现正弦波发生器时，先要对正弦波形模拟电压进行离散化，如图7.13所示。对于一个正弦波取 N 等分的离散点，按定义计算出对应于1、2、3、 \dots 、 N 各离散点的数据值 D_1 、 D_2 、 D_3 、 \dots 、 D_N 制成一个正弦表。因为正弦波在半周期内是以极值点为中心对称的，而且正、负波形为互补关系，故在制正弦表时只需进行1/4周期，即取 $0 \sim \pi/2$ 之间的数值，步骤如下：

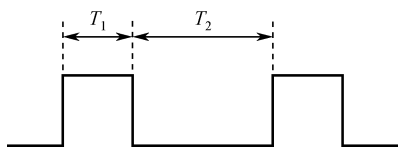


图7.12 矩形波

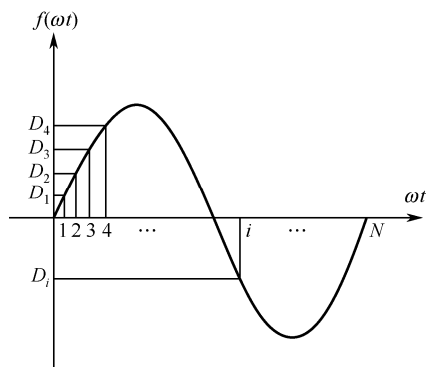


图7.13 正弦波形的离散化

- 计算 $0 \sim \pi/2$ 区间 $N/4$ 个离散的正弦值；
- 根据对称关系，复制 $\pi/2 \sim \pi$ 区间的值；
- 将 $0 \sim \pi$ 区间各点根据求补即得 $\pi \sim 2\pi$ 区间各值。

将得到的这些数据根据所用DAC的位数进行量化，得到相应的数字值，依次存入RAM中或固化在EPROM中，从而得到一个全周期的正弦编码表。

The diagram illustrates a microcontroller-based signal processing circuit. The 51-series microcontroller (U2) is configured with a crystal oscillator (X1) and capacitors (C2, C3) for timing. It interfaces with a DAC0832 (U1) to convert digital data into an analog signal. This signal is then amplified by an op-amp (U3), with its output waveform displayed on a graph. Additionally, a 74LS21 (U4) is used to generate various waveforms (staircase, triangle, square, and sine waves) from the microcontroller's digital outputs.

例7-6 采用DAC0832实现波形发生器的C51程序。



```

/***** 阶梯波函数 *****/
void st() {
    uchar i=0;
    while (KST) {
        XBYTE[DAC]=i++;    //启动DAC
    }
}

/***** 三角波函数 *****/
void tri() {
    uchar i=0;
    XBYTE[DAC]=i;    //启动DAC
    do {
        XBYTE[DAC]=i;    //上升沿
        i++;
    } while (i<0xff);
    do {
        XBYTE[DAC]=i;    //下降沿
        i--;
    } while (i>0x0);
}

/***** 方波函数 *****/
void sq() {
    XBYTE[DAC]=0x00;    //启动DAC
    delay();
    XBYTE[DAC]=0xff;
    delay();
}

/***** 正弦波函数 *****/
void sin() {
    uchar i;
    for(i=0;i<18;i++) XBYTE[DAC]=SINTAB[i];    //第一个1/4周期
    for(i=18;i>0;i--) XBYTE[DAC]=SINTAB[i];    //第二个1/4周期
    for(i=0;i<18;i++) XBYTE[DAC]=~SINTAB[i];    //第三个1/4周期
    for(i=18;i>0;i--) XBYTE[DAC]=~SINTAB[i];    //第四个1/4周期
}

/***** 主函数 *****/
main() {
    EX0=1;IT0=1;EA=1;
    while(1) {

```

```

        if(KST==1) st();
        if(KTRI==1) tri();
        if(KSQ==1) sq();
        if(KSIN==1) sin();
    }
}

/***** INT0中断服务函数 *****/
int0() interrupt 0 using 1{
    if(K1==0){                //判阶梯波键是否按下
        Tbase=0;
        KST=1;
    }
    if(K2==0){                //判三角波键是否按下
        Tbase=0;
        KTRI=1;
    }
    if(K3==0){                //判方波键是否按下
        Tbase=0;
        KSQ=1;
    }
    if(K4==0){                //判正弦波键是否按下
        Tbase=0;
        KSIN=1;
    }
}
}

```

采用程序软件控制DAC可以做成任意的波形发生器。凡是用数学公式可以表达的曲线，或无法用数学公式表达但可以画出来的曲线，都可以用计算机在DAC接口上复制出来。图7.15为任意波形的离散化。离散时取的采样点越多，数值量化的位数越多，则用DAC复现的波形精度越高。当然这时会在复现速度和内存方面付出代价。在程序控制下的波形发生器可以对波形的幅值标度和时间轴标度进行扩展或压缩及对数转换，因而应用十分方便。

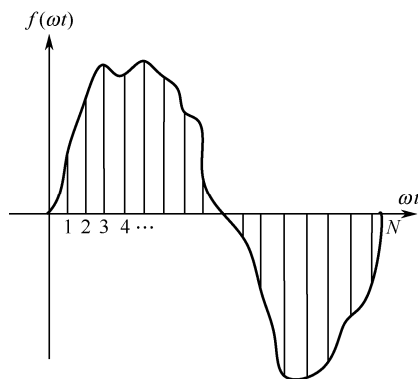


图7.15 任意波形的离散化

7.3 模/数转换器ADC接口技术

模/数转换器ADC (Analog-Digital Converter) 的功能是将输入模拟量转换为与其成比例的数字量, 是单片机应用系统的一种重要组成器件, 按其工作原理, 有比较式ADC、积分式ADC及电荷平衡(电压—频率转换)式ADC等。表7.3所列几种常用A/D芯片的特点和性能。表中带*号的为积分型ADC芯片, 其余均为比较式ADC芯片。

表7.3 几种常用A/D芯片的特点和性能

芯片型号	分辨率 (位数)	转换时间	转换误差	模拟输入范围	数字输出电平	外部 时钟	工作电 压	基准电压 (VREF)
ADC0801,0802, 0803、0804	8位	100 μ s	$\pm 1/2 \sim$ $\pm 1\text{LSB}$	0 \sim +5V	TTL电平	可以 不要	单电源 +5V	可不外接或 V_{REF} 为 1/2量程值
ADC0808、 0809	8位	100 μ s	$\pm 1/2 \sim$ $\pm 1\text{LSB}$	0 \sim +5V	TTL电平	要	单电源 +5V	$V_{\text{REF}}(+)\leq V_{\text{CC}}$ $V_{\text{REF}}(+)\geq 0$
ADC1210	12位 或10位	100 μ s(12位) 30 μ s(10位)	$\pm 1/2\text{LSB}$	0 \sim +5V 0 \sim +10V -5 \sim +5V	CMOS电平 (由 V_{REF} 决定)	要	+5V \sim $\pm 15\text{V}$	+5V或+15V
AD574	12位 或8位	25 μ s	$\pm 1\text{LSB}$	0 \sim +10V 0 \sim +20V -5 \sim +5V -10 \sim +10V	TTL电平	不要	$\pm 15\text{V}$ 或 $\pm 12\text{V}$ 和+5V	不需外供
*7109	12位	$\geq 30\text{ms}$	$\pm 2\text{LSB}$	-4 \sim +4V	TTL电平	可以 不要	+5V 和-5V	V_{REF} 为1/2量程值
*14433	3位半	$\geq 100\text{ms}$	$\pm 1\text{LSB}$	-0.2V \sim +0.2V -2 \sim +2V	TTL电平	可以 不要	+5V 和-5V	V_{REF} 为量程值
*7135	4位半	100ms左右	$\pm 1\text{LSB}$	-2 \sim +2V	TTL电平	要	+5V 和-5V	V_{REF} 为1/2量程值

在实际使用中, 应根据具体情况选用合适的ADC芯片。例如, 某测温系统的输入范围为0 \sim 500 $^{\circ}\text{C}$, 要求测温的分辨率为2.5 $^{\circ}\text{C}$, 转换时间在1ms之内, 可选用分辨率为8位的逐次比较式ADC0808/0809芯片。如果要求测温的分辨率为0.5 $^{\circ}\text{C}$ (即满量程的1/1000), 转换时间为0.5s, 则可选用双积分型ADC芯片7135。不同的芯片具有不同的连接方式, 其中最主要的是输入、输出及控制信号的连接方式。从输入端来看, 有单端输入的, 也有差动输入的。差动输入有利于克服共模干扰。输入信号的极性有单极性和双极性, 由极性控制端的接法决定, 从输出方式来看, 主要有两种:

- ① 数据输出寄存器具有可控的三态门, 此时芯片输出线允许和CPU的数据总线直接相连, 并在转换结束后, 利用读信号 $\overline{\text{RD}}$ 控制三态门将数据送上总线。
- ② 不具备可控的三态门, 输出寄存器直接与芯片引脚相连, 此时芯片的输出线必须通过输入缓冲器连至CPU的数据总线。

ADC芯片的启动转换信号有电平和脉冲两种形式, 设计时应特别注意, 对要求用电平启

动转换的芯片，如果在转换过程中撤去电平信号，则芯片将停止转换而得到错误的结果。

ADC转换完成后，将发出结束信号，以示主机可以从转换器读取数据。结束信号也用来向CPU发出中断申请，CPU响应中断后，在中断服务子程序中读取数据，也可用延时等待和查询转换是否结束的方法来读取数据。

7.3.1 比较式ADC0809接口应用编程

图7.16为阶梯波比较式ADC的工作原理。转换开始时，计数器复0，ADC的输出为 $V_d=0$ 。若输入电压 V_i 为正，则比较器输出 V_c 为正，与门打开，计数器对时钟脉冲进行计数，ADC输出即随计数脉冲的增加而增加，如图7.16(b)所示，当 $V_d > V_i$ 时，比较器输出变负，与门关闭，停止计数。计数器的计数值正比于输入电压，完成了从输入模拟量—电压到计数器的计数值—数字量的转换。

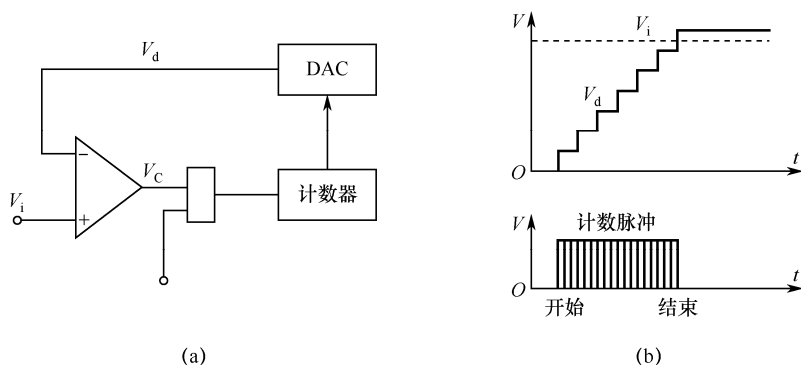


图7.16 阶梯波比较式ADC的工作原理

ADC0809是一种较为常用的8路模拟量输入、8位数字量输出的逐次比较式ADC芯片。图7.17为ADC0809的原理结构框图。芯片的主要部分是一个8位的逐次比较式A/D转换器。为了能够实现8路模拟信号的分时采集，在芯片内部设置了多路模拟开关及通道地址

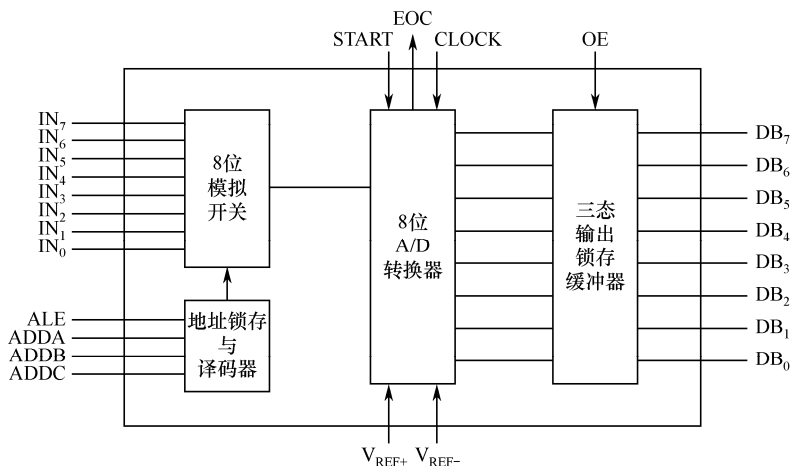


图7.17 ADC0809的原理结构框图

锁存和译码电路,因此能对多路模拟信号进行分时采集和转换。转换后的数据送入三态输出数据锁存器。ADC0808/0809的最大不可调误差为 $\pm 1\text{LSB}$,典型时钟频率为 640kHz ,时钟信号应由外部提供。每一个通道的转换时间约为 $100\mu\text{s}$ 。图7.18为ADC0809的引脚排列图,各引脚功能见表7.4。

图7.19为ADC0809的工作时序。由于ADC0809芯片没有专门的片选端,因此在设计与单片机接口时必须参考其工作时序。

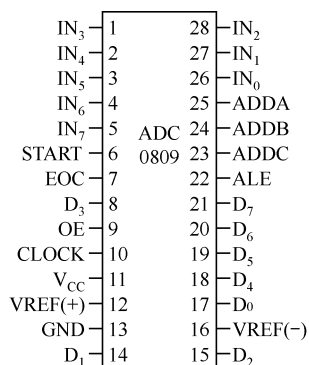


图7.18 ADC0809的引脚排列图

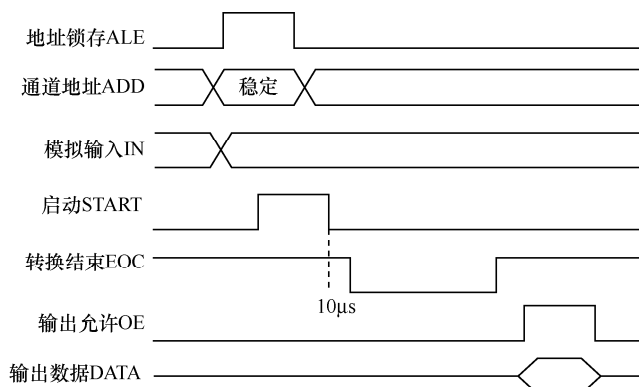


图7.19 ADC0809的工作时序

表7.4 ADC0809的各引脚功能

引 脚	功 能
$\text{IN}_0 \sim \text{IN}_7$	8路模拟量输入端
$\text{D}_0 \sim \text{D}_7$	数字量输出端
START	启动脉冲输入端,脉冲上升沿复位0809,下降沿启动A/D转换
ALE	地址锁存信号,高电平有效时把三个地址信号送入地址锁存器,并经地址译码得到地址输出,用以选择相应的模拟输入通道
EOC	转换结束信号,转换开始时变低,转换结束时变高,变高时将转换结果打入三态输出锁存器。如果将EOC和START相连,加上一个启动脉冲则连续进行转换
OE	输出允许信号输入端
CLOCK	时钟输入信号,最高允许值为 640kHz
VREF (+)	正基准电压输入端
VREF (-)	负基准电压输入端。通常将VREF (+)接+5V, VREF (-)接地
V_{cc}	电源电压,可从+5~+15V

图7.20为ADC0809与单片机8051的中断方式接口电路。采用线选法规定其端口地址,用单片机的P2.7引脚作为片选信号,因此端口地址为 7FFFH 。片选信号和 $\overline{\text{WR}}$ 信号一起经或非门产生ADC0809的启动信号START和地址锁存信号ALE;片选信号和 $\overline{\text{RD}}$ 信号一起经或非门产生ADC0809输出允许信号OE, $\text{OE}=1$ 时选通三态门使输出锁存器中的转换结果送入数据总线。ADC0809的EOC信号经反相后接到8051的 $\overline{\text{INT1}}$ 引脚用于产生转换完成的中断请求信号。ADC0809芯片的3位模拟量输入通道地址码输入端A、B、C分别接到8051的P0.0、P0.1和P0.2,故只要向端口地址 7FFFH 分别写入数据 $00\text{H} \sim 07\text{H}$,即可启动模拟量输入通道0~7进行A/D转换。

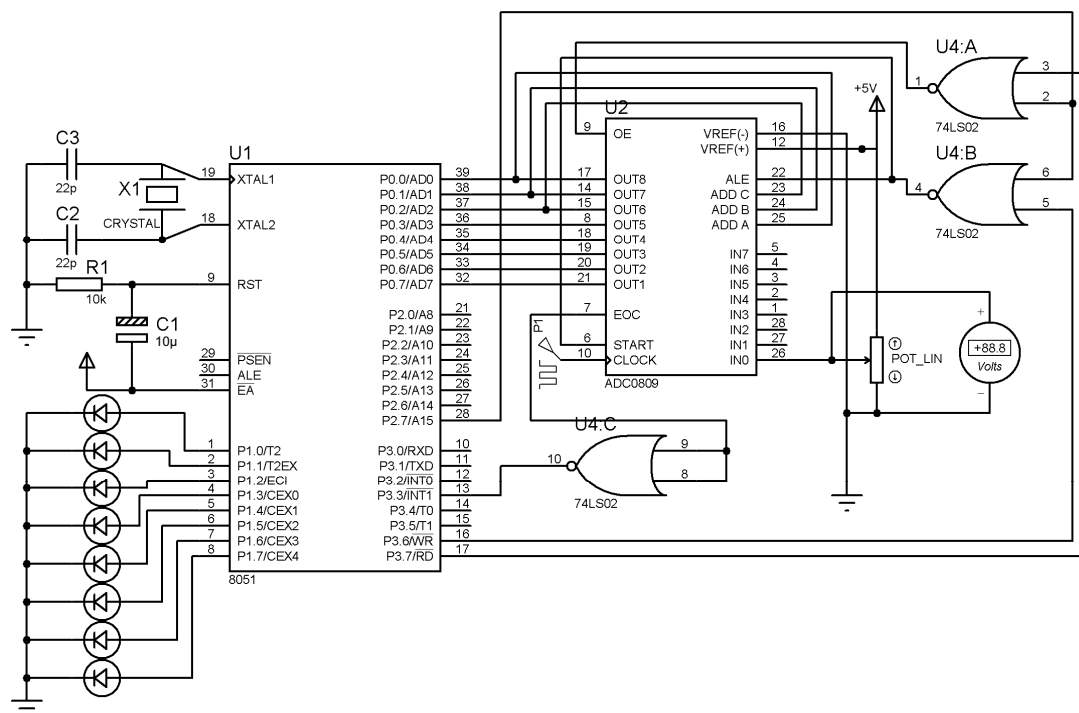


图7.20 ADC0809与单片机8051的中断方式接口电路

例7-7为中断工作方式下对8路模拟输入信号依次进行A/D转换的C51程序，8路输入信号的转换结果存储在内部数据存储单元内，并将第0路转换结果送到P1口显示。

例7-7 中断方式下8路模拟量输入的A/D转换C51程序。

```
#include <reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define ADC 0x7fff //定义ADC0809端口地址

uchar data ADCDat[8] _at_ 0x30;
uchar i=0;

/***** 主函数 *****/
main() {
    EX1=1; IT1=1; EA=1;
    XBYTE[ADC]=i; //启动ADC0808第0通道
    while(1) {
        P1=ADCDat[0]; //0通道转换结果送P1口显示
    }
}

/***** INT1中断服务函数 *****/
int1() interrupt 2 using 1 {
    ADCDat[i]=XBYTE[ADC]; //读取ADC0808转换结果
}
```

```

i++;
XBYTE[ADC]=i;          //启动ADC0808下一通道
if(i==8){
    i=0;
    XBYTE[ADC]=i;      //重新启动ADC0808第0通道
}
}

```

图7.21为ADC0809与单片机8051的查询方式接口电路。单片机8051的P2.7引脚作为片选信号，ADC0809芯片的3位模拟量输入通道地址码输入端A、B、C分别接到8051的最低3位地址线，因此8个输入通道的地址为7F00H~7F07H，需要分别对这8个地址进行写操作来启动A/D转换。ADC0809的EOC信号接到8051的P3.3引脚，通过查询P3.3引脚的电平状态判断A/D转换是否完成。

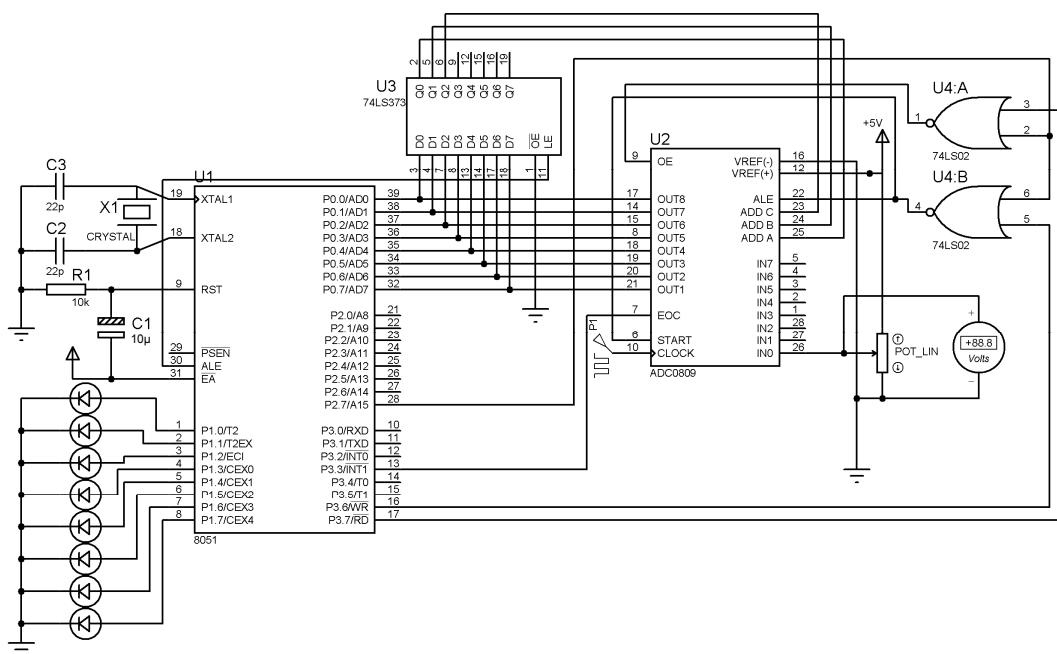


图7.21 ADC0809与单片机8051的查询方式接口电路

例7-8为查询工作方式对8路模拟量输入信号依次进行A/D转换的C51程序，执行后，启动8路模拟输入通道进行A/D转换，8路转换结果存储在内部数据存储器内，并将第0路转换结果送到P1口显示。

例7-8 查询方式下8路模拟输入A/D转换C51程序。

```

#include <reg52.h>
#include <absacc.h>
#define uchar unsigned char
#define uint unsigned int

uchar data ADCDat[8] _at_ 0x30;
uchar i=0;

```



```

uint ADC=0x7f00;    //定义ADC0808通道0地址
sbit EOC=P3^3;

/***** 读取ADC结果函数 *****/
void ADCread() {
    ADCDat[i]=XBYTE[ADC];    //读取ADC0808转换结果
    ADC++;i++;
    XBYTE[ADC]=i;            //启动ADC0808下一通道
    if(i==8){
        i=0; ADC=0x7f00;
        XBYTE[ADC]=i;        //重新启动ADC0808第0通道
    }
}

/***** 主函数 *****/
main() {
    XBYTE[ADC]=0x00;        //启动ADC0808第0通道
    while(1) {
        if(EOC==1) ADCread(); //根据EOC查询状态,读取ADC结果
        P1=ADCdat[0];        //0通道转换数据送P1口显示
    }
}

```

7.3.2 积分式ADC ICL7135接口应用编程

有些单片机应用系统要求能在工业现场使用,由于现场通常存在很强的干扰,如大功率电动机的磁场等,而被测信号往往是很微弱的直流信号,因此如果不能有效地抑制干扰,则测量结果很可能会失去意义。这时就可以考虑采用积分式的ADC了。下面先介绍双积分式ADC的工作原理,如图7.22所示。

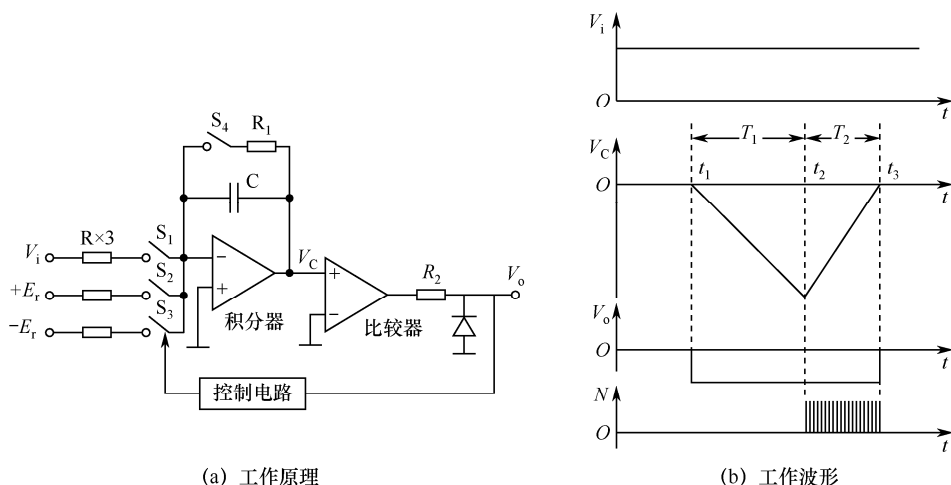


图7.22 双积分式ADC的工作原理及波形

工作过程分为三个阶段:

- 准备期: 开关 S_1 、 S_2 、 S_3 断开, S_4 接通, 积分电容 C 被短路, 输出为0。
- 采样期: 开关 S_2 、 S_3 、 S_4 断开, S_1 闭合, 积分器对输入模拟电压 V_i 进行积分, 积分时间固定为 T_1 , 在采样期结束的 t_2 时刻, 积分器输出电压为

$$V_c = -\frac{1}{RC} \int_{t_1}^{t_2} V_i dt = -\frac{T_1}{RC} \overline{V_i} \quad (7-1)$$

式中, $\overline{V_i} = \frac{1}{T_1} \int_{t_1}^{t_2} V_i dt$ 为被测模拟电压在 T_1 时间内的平均值。

- 比较期: 从 t_2 时刻开始, 开关 S_1 、 S_2 、 S_4 断开, S_3 闭合, 将与被测模拟电压极性相反的标准电压 $-E_r$ 接到积分器的输入端 (若被测模拟电压为 $-V_i$, 则 S_1 、 S_3 、 S_4 断开, S_2 闭合, 将 $+E_r$ 接到积分器的输入端), 使积分器进行反向积分。当积分器的输出回到0时, 比较器的输出发生跳变。设在 t_3 时刻积分器回0, 此时有

$$0 = V_c - \frac{1}{RC} \int_{t_2}^{t_3} (-E_r) dt = V_c + \frac{T_2}{RC} E_r \quad (7-2)$$

式中, $T_2 = t_3 - t_2$ 为比较周期。

将式 (7-1) 代入式 (7-2) 得

$$T_2 = \frac{T_1}{E_r} \overline{V_i} \quad (7-3)$$

在 T_2 周期内对一个周期为 τ 的时钟脉冲进行计数, 得

$$T_2 = N\tau \quad (7-4)$$

$$N = \frac{T_2}{\tau} = \frac{T_1}{\tau \times E_r} \overline{V_i} \quad (7-5)$$

由于 T_1 、 E_r 、 τ 都是恒定值, 从而计数值 N 就正比于被测模拟电压值, 实现A/D转换。双积分式ADC的采样周期通 T_1 常设计成对称干扰信号周期的整数倍, 以期对干扰信号有足够大的抑制能力, 如将 T_1 设计为工频信号的整数倍, 将对工频干扰有非常高的抑制能力。

随着大规模集成电路工艺的发展, 目前已有多种单芯片集成电路双积分A/D转换器供应市场, 如MC14433、ICL7135等。

ICL7135是一种常用的4位半BCD码双积分型单片集成ADC芯片, 分辨率相当于14位二进制数, 转换精度高, 转换误差为 $\pm 1\text{LSB}$, 并且能在单极性参考电压下对双极性输入模拟电压进行A/D转换, 模拟输入电压范围为 $0 \sim \pm 1.9999\text{V}$ 。芯片采用了自动校零技术, 可保证零点在常温下的长期稳定性, 模拟输入可以是差动信号, 输入阻抗极高。ICL7135芯片的引脚排列如图7.23所示。

ICL7135各引脚的功能见表7.5。

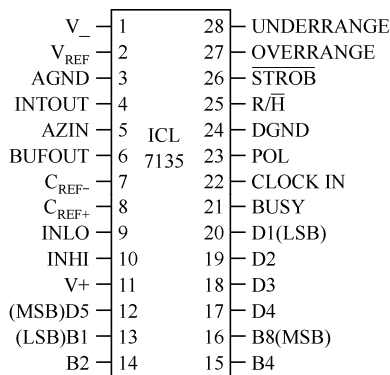


图 7.23 ICL7135 芯片的引脚排列

表7.5 ICL7135各引脚的功能

引 脚	功 能
V-	负电源端
V _{REF}	外接基准电压输入端
AGND	模拟地
INTOUT	积分器输出, 外接积分电容C _{INT} 端
AZIN	外接调零电容C _{AZ} 端
BUFOUT	缓冲器输出, 外接积分电阻R _{INT} 端
C _{REF-} 、C _{REF+}	外接基准电压电容C _{REF} 端, 电容值可取1μF
INLO、INHI	被测模拟电压(低、高)差分输入端, 单端输入时, INLO与通常模拟地连在一起
V+	正电源端
D5~D1	位扫描选通信号输出端, 其中D5对应万位, 其余依次为千、百、十、个位
B8~B1	BCD码输出端, 其中B ₈ 为最高位, B ₁ 为最低位
UNDERRANGE	欠量程标志输出端, 当输入信号小于量程的9%时, 该端输出高电平
OVERRANGE	过量程标志输出端, 当输入信号超过转换器计数范围(19999)时, 该端输出高电平
$\overline{\text{STROBE}}$	选通脉冲输出端, A/D转换结束后, 该端输出5个负脉冲, 分别选通从高位到低位的BCD码输出。 $\overline{\text{STROBE}}$ 也可作为中断请求信号, 向主机申请中断
R/ $\overline{\text{H}}$	A/D转换启动控制端, 该端接高电平时, ICL7135连续自动转换; 该端接低电平时, 转换结束后保持转换结果
DGND	数字地
POL	极性输出端, 输入信号为正时输出高电平, 输入信号为负输出低电平
CLOCK IN	时钟输入端, 输入信号为双极性时, 时钟最高频率为125kHz, 这时转换速率为3次/秒左右。如果输入信号为单极性, 则时钟频率可增加到1MHz, 这时转换速率为25次/秒左右
BUSY	输出状态信号端, 积分器在对输入信号的积分过程中, BUSY输出高电平(转换正在进行), 积分器反向积分过零后, BUSY输出低电平(转换已经结束)

ICL7135芯片工作时需要外接积分电阻、电容及调零电容。积分电阻 R_{INT} 的计算公式为

$$R_{\text{INT}} = \text{满度电压} / 20\mu\text{A} \quad (7-6)$$

积分电容 C_{INT} 的计算公式为

$$C_{\text{INT}} = \frac{10000 \times (1/f_{\text{osc}}) \times 20\mu\text{A}}{\text{积分器输出摆幅}} \quad (7-7)$$

如果电源电压取 $\pm 5\text{V}$, 电路的模拟地端接0V, 则积分器输出摆幅取 $\pm 4\text{V}$ 较为合适。调零电容 C_{AZ} 可取1μF。为了使ICL7135工作在最佳状态, 获得最好的性能, 必须注意外接元器件的选择。

图7.24为ICL7135的输出时序图。

ICL7135的转换结果输出是动态的, 因此必须通过并行接口才能与单片机连接。图7.25为ICL7135与单片机8051的接口电路。图中, 74LS157为4位2选1的数据多路开关, $\overline{\text{A}}/\text{B}$ 端输入为低电平时, 1A、2A、3A输入信息在1Y、2Y、3Y输出; $\overline{\text{A}}/\text{B}$ 端为高电平时, 1B、2B、3B输入信息在1Y、2Y、3Y输出。因此, 当ICL7135的高位选通信号D5输出为高电平时, 万位数据B1和极性、过量程、欠量程标志输入到单片机8051的P0.0~P0.3, 当D5为低电平时, ICL7135的B8、B4、B2、B1输出低位转换结果的BCD码, 此时BCD码数据线B8、B4、B2、B1输入到单片机8051的P0.0~P0.3。

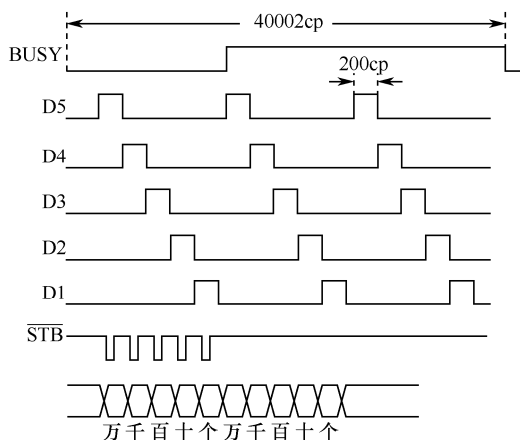


图7.24 ICL7135的输出时序图

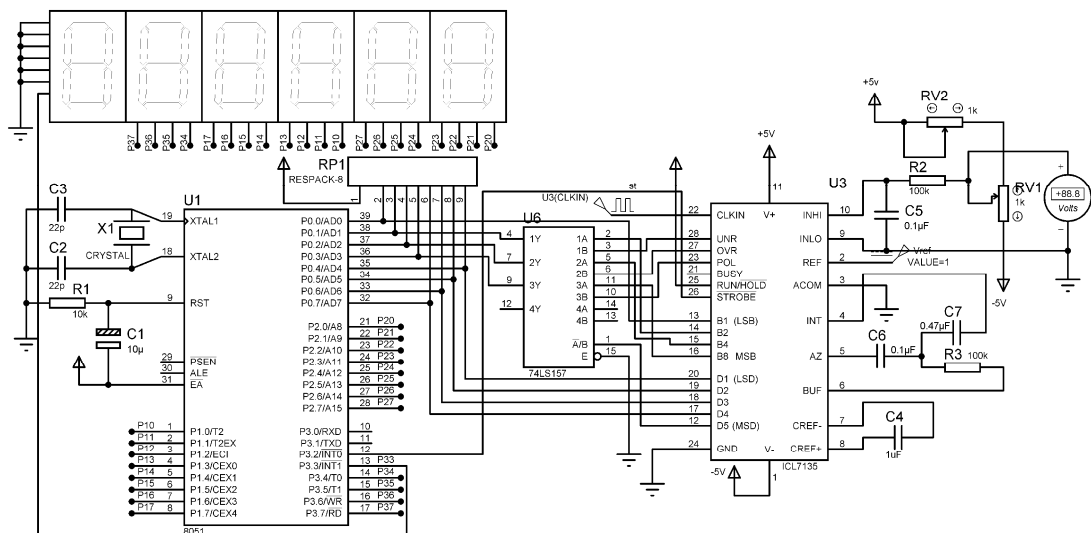


图7.25 ICL7135与单片机8051的接口电路

ICL7135的时钟频率为125kHz，每秒进行3次A/D转换。ICL7135的数据输出选通脉冲线 STROBE 接到单片机外部中断INT0端，当ICL7135完成一次A/D转换后，产生5个数据选通脉冲，分别将各位的BCD码结果和标志D1~D5打入8051的P0口。由于ICL7135的A/D转换是自动进行的，因此在完成一次A/D转换后，选通脉冲的产生和8051中断的开放是不同步的。为了保证读出数据的完整性，单片机只对最高位（万位）的中断请求做出响应，低位数据的输入则采用查询的方法，A/D转换结果送入单片机片内RAM的20H、21H和22H单元，数据存放格式为

D7	D6	D5	D4	D3	D2	D1	D0
POL	OV	UN		万	位		
D7	D6	D5	D4	D3	D2	D1	D0
千	位			百	位		
D7	D6	D5	D4	D3	D2	D1	D0
十	位			个	位		

例7-9为采用C51编写的ICL7135A/D转换及数据显示程序。主程序完成开中断等初始化工作后，进入查询等待ICL7135完成一次A/D转换的结果标志。中断服务程序读取一次完整的A/D转换结果后，置“1”标志位PSW.5，主程序通过查询该标志位的状态，将BCD码的结果数据通过单片机的I/O端口送到数码管显示。

例7-9 ICL7135A/D转换及数据显示C51程序。

```
#include<reg52.h>
#include <absacc.h>
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int

uchar data ADCDat[3] _at_ 0x20;
uchar bdata ADCbase _at_ 0x2f;

sbit ADC=ADCbase^5;
sbit COM=P3^3;

/***** 数据显示函数 *****/
void Tran(){
    uchar Dat;
    Dat=ADCDat[0];
    if((Dat&0x40)==0x40){
        P1=0xFF;P2=0xFF;P3|=0xF0;
        return;
    }
    if((Dat&0x20)==0x20){
        P1=0x00;P2=0x00;P3&=0x0F;
        return;
    }
    if((Dat&0x80)==0x80) COM=0;
    else COM=1;
    Dat=_crol_(ADCDat[0],4);
    Dat=Dat&0xf0;P3&=0x0f;P3|=Dat;
    P1=ADCDat[1];P2=ADCDat[2];
    return;
}

/***** 主函数 *****/
void main(){
    P0=0xFF;TCON=0x01;IE=0x81; //初始化，开中断
    while(1){
        if(ADC==1){ //查询ICL7135 A/D转换完成标志
            ADC=0;Tran(); //显示A/D转换结果数据
        }
    }
}
```

```

/***** ICL7135中断服务程序 *****/
void int0() interrupt 0 using 1{
    uchar Dat1;
    IE=0x00;           //关中断
    Dat1=P0;           //读取A/D转换结果的万位数据
    if((Dat1&0xf0)==0){ //判断D5
        Dat1=_crol_((Dat1&0x0f),4);
        ADCDat[0]=(Dat1&0xe0)|(_crol_(Dat1,4)&0x01);
        do Dat1=P0;    //读取A/D转换结果的千位数据
            while((Dat1&0x80)==0);
        ADCDat[1]=_crol_((Dat1&0x0f),4);
        do Dat1=P0;    //读取A/D转换结果的百位数据
            while((Dat1&0x40)==0);
        Dat1=Dat1&0x0f;
        ADCDat[1]=ADCDat[1]|Dat1;
        do Dat1=P0;    //读取A/D转换结果的十位数据
            while((Dat1&0x20)==0);
        ADCDat[2]=_crol_((Dat1&0x0f),4);
        do Dat1=P0;    //读取A/D转换结果的个位数据
            while((Dat1&0x10)==0);
        Dat1=Dat1&0x0f;
        ADCDat[2]=ADCDat[2]|Dat1;
        ADC=1;
    }
    IE=0x81;           //开中断
}

```

7.3.3 串行A/D芯片TLC549接口应用编程

前面介绍了比较式和积分式的ADC接口技术，它们都属于并行接口方式，电路结构较为复杂。为了简化接口电路，许多半导体厂商推出了串行接口的ADC芯片。美国TI公司推出的TLC549就是一种常用的低功耗8位串行ADC芯片。TLC549具有4MHz片内系统时钟和软、硬件控制电路，转换时间最长为17μs，最高转换速率为40000次/s，总失调误差最大为±0.5LSB，典型功耗值为6mW。采用差分参考电压高阻输入，抗干扰，可按比例量程校准转换范围。

TLC549的极限参数如下：

- 电源电压：6.5V。
- 输入电压范围：0.3V~VCC+0.3V。
- 输出电压范围：0.3V~VCC+0.3V。
- 峰值输入电流（任一输入端）：±10mA。
- 总峰值输入电流（所有输入端）：±30mA。
- 工作温度：0℃~70℃。

图7.26为TLC549的内部结构框图和引脚排列，各引脚功能见表7.6。

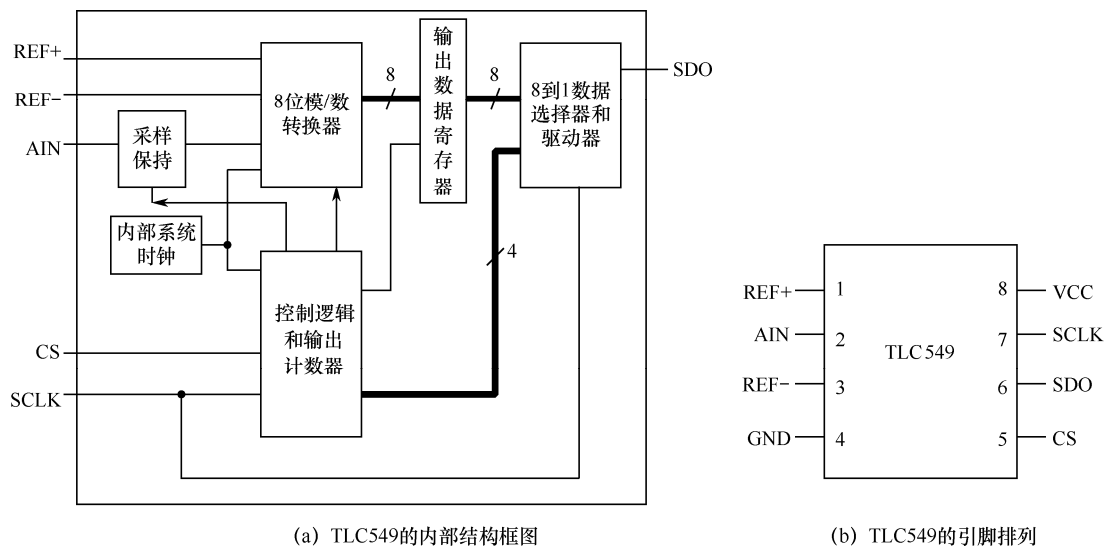


图7.26 TLC549的内部结构框图和引脚排列

表7.6 TLC549各引脚功能

引 脚	功 能
REF+、REF-	基准电压正、负端
AIN	模拟量串行输入端
GND	接地端
\overline{CS}	片选端，低电平有效
SDO	数字量输出端
SCLK	时钟信号端
VCC	电源端

TLC549的工作时序如图7.27所示。当 \overline{CS} 为高电平时，数据输出端（SDO）处于高阻状态，此时SCLK不起作用。这种 \overline{CS} 控制作用允许在同时使用多片TLC549时公用SCLK，以减少多路A/D并用时的I/O控制端口。在通常情况下， \overline{CS} 应为低电平。

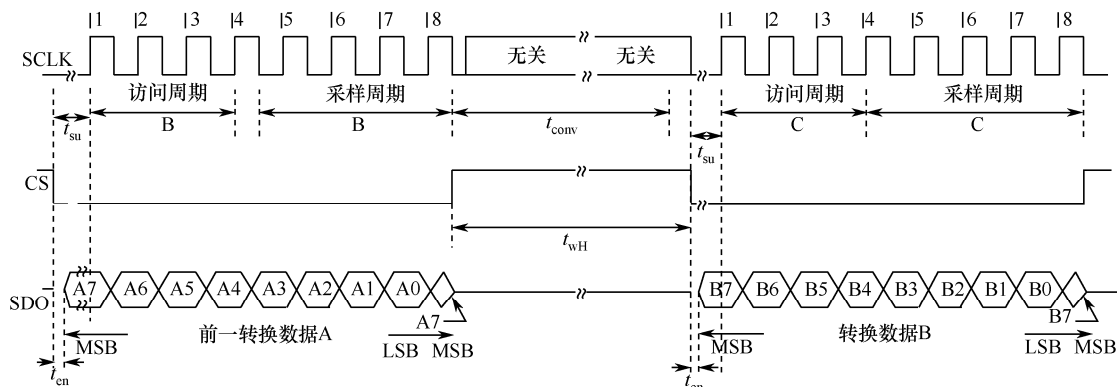


图7.27 TLC549的工作时序


```
Dat=0;
CS=0;
for(i=0;i<8;i++){
    SCLK=1;
    Dat<<=1;           //获得转换数据
    if(SDO) Dat|=1;
    SCLK=0;
}
CS=1;                 //转换数据送P1口显示
P1=Dat;
}
/***** 主函数 *****/
void main(){
    uchar i;
    while(1){
        TLC549();      //启动A/D转换
        for(i=0;i<200;i++) //延时
            _nop_();
    }
}
```


I²C总线接口应用

8.1 I²C总线简介

I²C总线是一种简单、双向二线制同步串行总线，只需要两根线（串行时钟线和串行数据线）即可在连接于总线上的器件之间传送信息。这种总线的主要特性如下：

- ① 只有两根线，串行时钟线和串行数据线；
- ② 每个连到总线上的器件都可由软件以唯一的地址寻址，并建立简单的主/从关系；
- ③ 主器件既可作为发送器，也可作为接收器；
- ④ 一个真正的多主总线，带有竞争检测和仲裁电路，可使多个主机任意同时发送数据而不破坏总线上的数据信息；
- ⑤ 同步时钟允许器件通过总线以不同的波特率进行通信；
- ⑥ 同步时钟可以作为停止和重新启动串行口发送的握手方式；
- ⑦ 连接到同一总线上的集成电路器件数只受400pF的最大总线电容的限制。

I²C总线极大地方便了系统设计者，无须设计总线接口，因为总线接口已经集成在芯片内了，从而使设计时间大为缩短，并且从系统中移去或增加集成电路芯片对总线上的其他集成电路芯片没有影响。I²C总线的简单结构便于产品改型或升级。改型或升级时只需从总线上取消或增加相应的集成电路芯片即可。

目前，市场上有包括LED驱动器、LCD驱动器、A/D和D/A转换器、RAM、EPROM及I/O接口等在内的上百种I²C接口电路芯片。对于原来没有I²C总线的单片机，可以采用I²C接口扩展器件PCD8584等扩展出I²C总线接口，或者采用软件程序模拟I²C总线的时序来完成接口功能。

8.2 I²C总线结构与数据传输

I²C总线接口的电气结构如图8.1所示。组成I²C总线的串行数据线SDA和串行时钟线SCL必须经过上拉电阻R_p接到正电源上，连接到总线上器件的输出级必须为“开漏”或“开集”

形式，以便完成“线与”的功能。

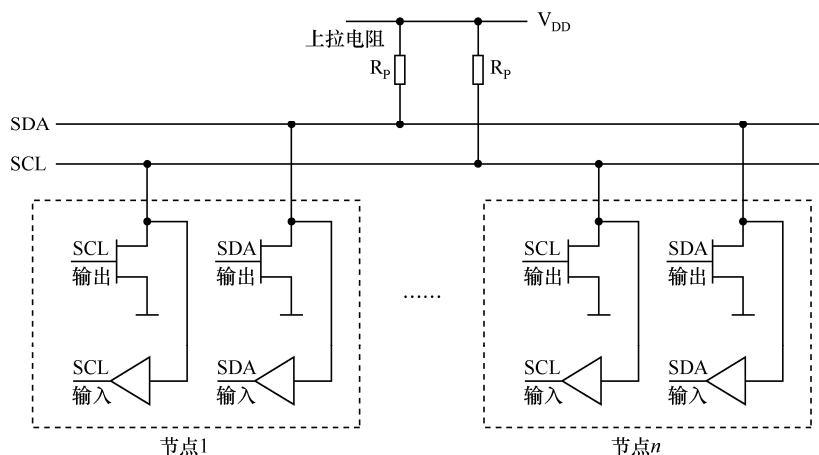


图8.1 I²C总线接口的电气结构

I²C总线上可以实现多主双向同步数据传送，所有主器件都可发出同步时钟，但由于SCL接口的“线与”结构，一旦一个主器件时钟跳变为低电平，将使SCL线保持为低电平直至时钟达到高电平，因此SCL线上时钟低电平时间由各器件中时钟最长的低电平时间决定，而时钟高电平时间则由高电平时间最短的器件决定。

为了使多主数据传送能够正确实现，I²C总线中带有竞争检测和仲裁电路。总线竞争的仲裁和处理由内部硬件电路来完成。当两个主器件发送数据相同时不会出现总线竞争；当两个主器件发送数据不同时才出现总线竞争。总线竞争的仲裁过程如图8.2所示。

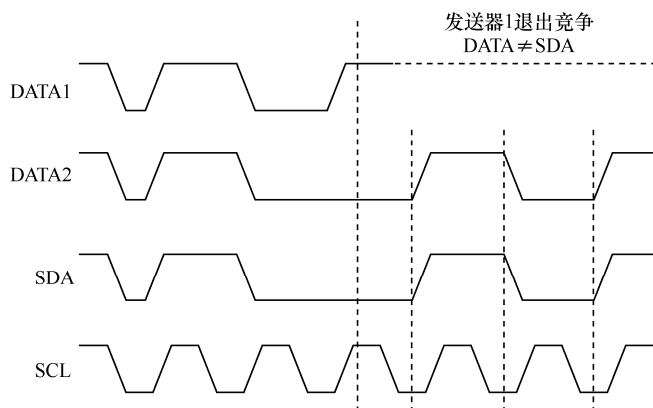


图8.2 总线竞争的仲裁过程

当某一时刻主器件1发送高电平而主器件2发送低电平时，由于SDA的“线与”作用，主器件1发送的高电平在SDA线上反映的是主器件2的低电平状态，因此这个低电平状态通过硬件系统反馈到数据寄存器中，因与原有状态比较不同而退出竞争。

I²C总线可以构成多主数据传送系统，但只有带CPU的器件可以成为主器件。主器件发送时钟、启动位、数据工作方式，从器件则接收时钟及数据工作方式。接收或发送则根据数据的传送方向决定。I²C总线上数据传送时的启动、结束和有效状态都由SDA、SCL的电平状

态决定，在I²C总线规约中启动和停止条件规定如下。

启动条件：在SCL为高电平时，SDA出现一个下降沿则启动I²C总线。

停止条件：在SCL为高电平时，SDA出现一个上升沿则停止使用I²C总线。

除了启动和停止状态，在其余状态下，SCL的高电平都对应于SDA的稳定数据状态。每一个被传送的数据位由SDA线上的高、低电平表示，对于每一个被传送的数据位都在SCL线上产生一个时钟脉冲。在时钟脉冲为高电平期间，SDA线上的数据必须稳定，否则被认为是控制信号。SDA只能在时钟脉冲SCL为低电平期间改变。启动条件后，总线为“忙”，在结束信号过后的一定时间内，总线被认为是“空闲”的。

在启动和停止条件之间可传送的数据不受限制，但每个字节必须为8位。首先传送最高位，采用串行传送方式，但在每个字节之后必须跟一个响应位。主器件收发每个字节后产生一个时钟应答脉冲，在这期间，发送器必须保证SDA为高，由接收器将SDA拉低，称为应答信号（ACK）。主器件为接收器时，在接收了最后一个字节之后不发应答信号，也称为非应答信号（NOT ACK）。当从器件不能再接收另外的字节时也会出现这种情况。I²C总线的数据传输时序如图8.3所示。

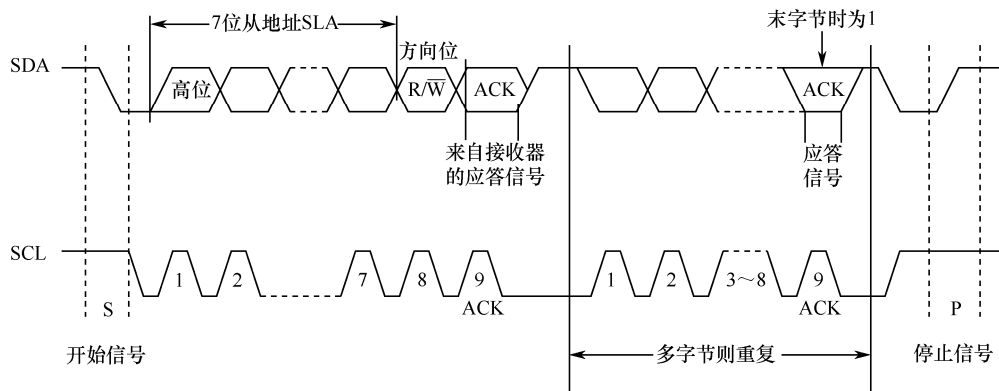


图8.3 I²C总线上的数据传输时序

I²C总线中每个器件都有自己唯一确定的地址，启动条件后，主机发送的第一个字节就是被读/写的从器件地址。其中，第8位为方向位，“0”（W）表示主器件发送，“1”（R）表示主器件接收。总线上每个器件在启动条件后都把自己的地址与前7位相比较，如相同，则器件被选中，产生应答，并根据读/写位决定在数据传送中是接收还是发送。无论是主发、主收还是从发、从收都是由主器件控制的。

在主发送方式下，由主器件先发出启动信号（S），接着发从器件的7位地址（SLA）和表明主器件发送的方向位0（W），即这个字节为SLA+W。被寻址的从器件在收到这个字节后返回一个应答信号（A）。在确定主从握手应答正常后，主器件向从器件发送字节数据，从器件每收到一个字节数据后都要返回一个应答信号，直到全部数据都发送完为止。

在主接收方式下，主器件先发出启动信号（S），接着发从器件的7位地址（SLA）和表明主器件接收的方向位1（R），即这个字节为SLA+R。在发送完这个字节后，SCL继续输出时钟，通过SDA接收从器件发来的串行数据。主器件每接收到一个字节后都要发送一个应答信号（A）。当全部数据都发送或接收完毕后，主器件应发出停止信号（P）。图8.4为主器件发送和接收数据的过程。

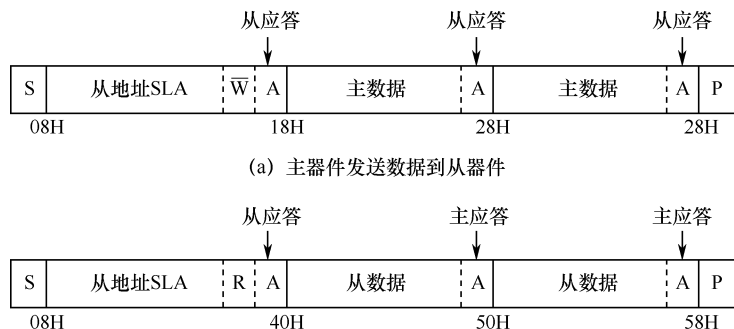
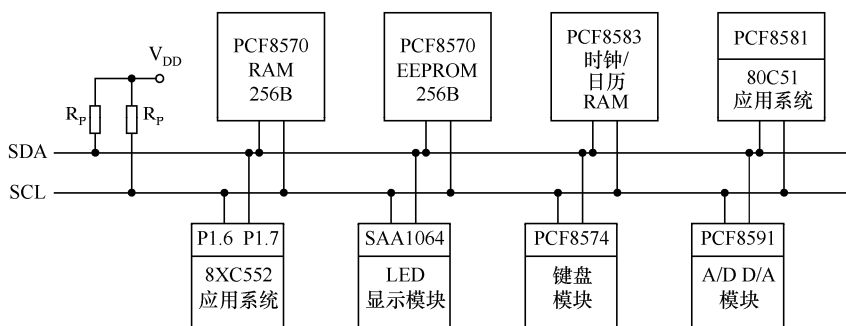


图8.4 主器件发送和接收数据的过程

典型的I²C总线应用系统结构如图8.5所示。I²C总线上可挂接 n 个单片机应用系统和 m 个带I²C接口的器件，每个I²C接口作为一个节点。节点的数量和种类主要受总电容量和地址容量的限制。单片机节点可编程为主器件或从器件，而器件节点则只能编程为从器件。8XC552单片机带有I²C接口，可以直接挂在I²C总线上，对于没有I²C接口的单片机，可通过I²C接口扩展芯片PCD8584扩展出I²C接口。I²C总线系统中的单片机原有的并行接口和异步通信接口资源可不受I²C总线限制任意扩展，I²C总线系统中的器件节点可构成各种标准功能模块。I²C总线上的所有节点都有约定的地址以便实现可靠的数据传送。单片机节点可作为主器件或从器件，作为主器件时其地址无意义，作为从器件时其从地址在初始化程序中定位在I²C总线地址寄存器S1ADR的高7位中。器件节点的7位地址由两部分组成，完全由硬件确定：一部分为器件编号地址，由芯片厂家规定；另一部分为引脚编号地址，由引脚的高、低电平决定，如四位LED驱动器SAA1064的地址为01110A1A0。其中，01110为器件编号地址，表明该器件为LED驱动器；A1A0为该器件的两个引脚，分别接高、低电平时可以有四片不同地址的LED驱动模块节点。256个字节EEPROM器件PCF8582的地址为1010A2A1A0，器件编号地址为1010，地址引脚有3个：A2A1A0，通过这3个引脚的不同电平设置，可连接8片不同地址的EEPROM芯片。芯片内地址则由主器件发送的第一个数据字节来选择。

图8.5 典型的I²C总线应用系统结构

I²C总线是一种串行通信总线，与并行总线不同。并行总线中有地址总线，CPU可通过地址总线来选择所需要器件的地址。I²C总线只有一根数据线和一根时钟线，没有专门的地址线，而是利用数据传送中的头几个字节来传送地址信息。I²C总线的寻址方式有主器件的节点寻址和通用呼叫寻址两种，具体实现方法是由主器件在发出启动位S后紧接着发送从器件的7位地址码，即S+SLA，在节点地址寻址中，SLA为被寻址的从节点地址，当SLA为全

“0”时，即为通用呼叫地址。通用呼叫地址用于寻址接到I²C总线上每个器件的地址，不需要从通用呼叫地址命令中获取数据的器件可以不响应通用呼叫地址。

8.3 I²C总线通用驱动程序

IAP15W4K58S4单片机没有内部硬件I²C总线，可以采用软件模拟的方法实现I²C总线接口功能。例8-1是一个采用C51编写的通用I²C总线通用驱动程序，可用于没有内部I²C硬件的单片机与I²C总线器件的接口。利用单片机的P1.6和P1.7引脚来模拟I²C总线SCL和SDA的工作时序，用户也可以定义其他I/O口引脚作为SCL和SDA信号。

例8-1 I²C总线通用驱动程序。

```
/* 全局符号定义 */
#define HIGH 1
#define LOW 0
#define FALSE 0
#define TRUE ~FALSE
#define uchar unsigned char

sbit SCL      = P1^6 ;
sbit SDA      = P1^7 ;

/***** 延时函数 *****/
void delay( void ) {
    ;
}

/***** I2C总线起始位函数 *****/
* 函数原型: void I_start(void);
* 功    能: 提供I2C总线工作时序中的起始位。
*****/
void I_start( void ) {
    SCL = HIGH ; delay() ;
    SDA = LOW  ; delay() ;
    SCL = LOW  ; delay() ;
}

/***** I2C总线停止位函数 *****/
void I_stop( void ) {
    SDA = LOW ; delay() ;
    SCL = HIGH ; delay() ;
    SDA = HIGH ; delay() ;
    SCL = LOW ; delay() ;
}

/***** I2C总线初始化函数 *****/
void I_init( void ) {
    SCL = LOW ;
```

```

    I_stop() ;
}

/***** I2C总线时钟信号函数 *****/
bit I_clock( void ) {
    bit sample ;
    SCL = HIGH ;
    delay() ;
    sample = SDA ;
    SCL = LOW ;
    delay() ;
    return ( sample ) ;
}

/***** I2C总线数据发送函数 *****/
bit I_send(uchar I_data ) {
    uchar I ;
    for ( I=0 ; I<8 ; I++ ) {          // 发送8位数据
        SDA = (bit)( I_data & 0x80 ) ;
        I_data = I_data << 1 ;
        I_clock() ;
    }
    SDA = HIGH ;                        // 请求应答信号ACK
    return ( ~I_clock() ) ;
}

/***** I2C总线数据接收函数 *****/
uchar I_receive( void ) {
    uchar I_data = 0 ;
    uchar I ;
    for ( I=0 ; I<8 ; I++ ) {
        I_data *= 2 ;
        if ( I_clock() )
            I_data++ ;
    }
    return ( I_data ) ;
}

/***** I2C总线应答函数 *****/
void I_Ack( void ) {
    SDA = LOW ;
    I_clock() ;
    SDA = HIGH ;
}

```

8.4 I²C接口存储器芯片24C04应用编程

24C04是一种I²C接口的EEPROM器件，具有512×8位的存储容量，工作在从器件方式，

每个字节可擦/写100万次，数据保存时间大于40年。写入时具有自动擦除功能、页写入功能，可一次写入16个字节。图8.6为24C04的引脚排列，各引脚功能见表8.1。

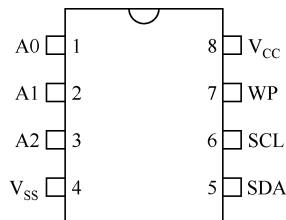


图8.6 24C04的引脚排列

表8.1 24C04各引脚功能

引 脚	功 能
A0、A1、A2	器件地址引脚，A1和A2决定芯片的从机地址，可接Vcc或Vss，A0为空引脚，不用连接
Vss	地
Vcc	正电源
WP	写保护，WP脚接Vcc时，禁止写入高位地址（100H~1FFH）；WP脚接Vss时，允许写入任何地址
SCL	时钟信号
SDA	串行数据输入端

单片机与24C04之间进行数据传递时，首先传送器件的从地址SLA，格式为

START	1	0	1	0	A2	A1	BA	R/W	ACK
-------	---	---	---	---	----	----	----	-----	-----

START为起始信号，1010为24C04器件地址，A2和A1由芯片的A2、A1引脚上的电平决定，这样可最多接入4片24C04芯片，BA为块地址（每块256字节），R/W决定是写入（0）还是读出（1），ACK为24C04给出的应答信号。在对24C04进行写入时，应先发出从机地址字节SLAW（R/W为0），再发出字节地址WORDADR和写入的数据data（可为1~16个字节），写入结束后，应发出停止信号。

通常对EEPROM器件写入时需要一定的写入时间（5~10ms），因此在写入程序中无法连续写入多个数据字节。为了解决连续写入多个数据字节的问题，EEPROM器件中常设有一定容量的页写入数据寄存器。用户一次写入EEPROM的数据字节不大于页写入字节数时，可按通常RAM的写入速度，将数据装入EEPROM的数据寄存器中，随后启动自动写入定时控制逻辑，经过5~10ms的时间，自动将数据寄存器中的数据同步写入EEPROM的指定单元。

这样一来，只要一次写入的字节数不多于页写入容量，总线对EEPROM的操作就可视为对静态RAM的操作，但要求下次数据写入操作在5~10ms之后进行。24C04的页写入字节数为16。对24C04进行页写入是指向其片内指定首地址（WORDADR）连续写入不多于 n 个字节数据的操作。 n 为页写入字节数， m 为写入字节数， $m \leq n$ 。页写入数据操作格式为

S	SLAW	A	WORDADR	A	data1	A	data2	A	...	datam	A	P
---	------	---	---------	---	-------	---	-------	---	-----	-------	---	---

这种数据写入操作实际上就是 $m+1$ 个字节的I²C总线进行主发送的数据操作。

对24C04写入数据时也可以按字节方式进行，即每次向其片内指定单元写入一个字节的数，这种写入方式的可靠性高。字节写入数据操作格式为

S	SLAW	A	WORDADR	A	data	A	P
---	------	---	---------	---	------	---	---

24C04的读操作与通常的SRAM相同，但每读一个字节，地址将自动加1。24C04有三种读操作方式，即现行地址读、指定地址读和序列读。

现行地址读是指不给定片内地址的读操作。读出的是现行地址中的数据。现行地址是片内地址寄存器当前的内容，每完成一个字节的读操作，地址自动加1，故现行地址是上次操作完成后的下一个地址。现行地址读操作时，应先发出从机地址字节SLAR（R/W为1），接收到应答信号（ACK）后即开始接收来自24C04的数据字节，每接收到一个字节的数据都必须发出一个应答信号（ACK）。现行地址读的数据操作格式为

S	SLAR	A	Data	A	P
---	------	---	------	---	---

指定地址读是指按指定的片内地址读出一个字节数据的操作。由于要写入片内指定地址，故应先发出从机地址字节SLAW（R/W为0），再进行一个片内字节地址的写入操作，最后发出重复起始信号和从机地址SLAR（R/W为1），开始接收来自24C04的数据字节。数据操作格式为

S	SLAW	A	WORDADR	A	S	SLAR	A	data	A	P
---	------	---	---------	---	---	------	---	------	---	---

序列读操作是指连续读入 m 个字节数据的操作。序列读入字节的首地址可以是现行地址或指定地址。其数据操作可以在上述两种操作的SLAR发送之后进行。数据操作格式为

S	SLAR	A	data1	A	data2	...	datam	A	P
---	------	---	-------	---	-------	-----	-------	---	---

图8.7为24C04与单片机的接口电路。

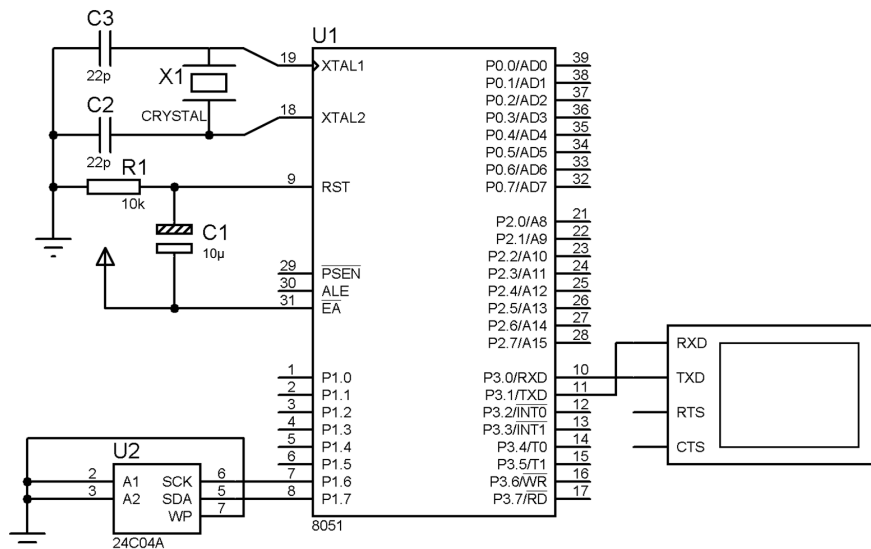


图8.7 24C04与单片机的接口电路

例8-2 单片机对24C04进行读/写操作的C51驱动程序。

```
#include <reg51.h>
#include <stdio.h>
```

```

#include <i2c.h>                                // 包含I2C总线通用驱动程序
#define WRITE 0xA0                             // 定义24C04的器件地址SLA和方向位W
#define READ 0xA1                             // 定义24C04的器件地址SLA和方向位R
#define BLOCK_SIZE 16                         // 定义指定字节个数
#define uchar unsigned char

uchar EAROMImage[16]="Hello everybody!";      // 定义写入数据
uchar transfer[16];                           // 定义数据单元

/***** 地址写入函数 *****/
* 功能：向24C04写入器件地址和一个指定的字节地址。
*****/
bit E_address(uchar Address ) {
    I_start() ;
    if ( I_send( WRITE ) )
        return ( I_send( Address ) ) ;
    else
        return ( FALSE ) ;
}

/***** 数据读取函数 *****/
* 功能：从24C04中读取BLOCK_SIZE个字节的数据并转存于8051片内RAM单元，
* 采用序列读操作方式连续读取数据。如果24C04不接受指定地址则返回0。
*****/
bit E_read_block( uchar start ) {
    uchar i ;
    if ( E_address( start ) ) { // 从指定地址开始读取数据
        I_start() ;           // 发送重复启动信号
        if ( I_send( READ ) ) {
            for ( i=0 ; i<=BLOCK_SIZE ; i++ ) {
                transfer[i]=(I_receive());
                if ( i != BLOCK_SIZE )
                    I_Ack() ;
            }
            else {
                I_clock() ;
                I_stop() ;
            }
            return ( TRUE ) ;
        }
        else {
            I_stop() ;
            return ( FALSE ) ;
        }
    }
}

```

```

    }
    else
        I_stop() ;
    return ( FALSE ) ;
}

/***** 5ms延时 函数 *****/
void wait_5ms( void ) {
    int I ;
    for ( I=0 ; I<1000 ; I++ ) {
        ;
    }
}

/***** 数据写入函数 *****/
* 功能：将数据写入到24C04指定地址开始的BLOCK_SIZE个字节。采用字节写方式，
* 每次写入时都需要指定片内地址。如果24C04不接受指定地址则返回0。
*****/
bit E_write_block( uchar start) {
    uchar i ;
    for ( i=0 ; i<=BLOCK_SIZE ; i++ ) {
        if ( E_address(i+start) && I_send( EAROMImage[i] ) ) {
            I_stop() ;
            wait_5ms();
        }
        else
            return ( FALSE ) ;
    }
    return ( TRUE ) ;
}

/***** 主函数 *****/
void main() {
    uchar addr = 0x50;           // 定义24C04片内地址
    SCON = 0x5a;
    TMOD = 0x20;
    TCON = 0x69;
    TH1 = 0xfd;
    WRITE = 0xA0;               // 定义24C04写入地址
    READ = 0xA1;               // 定义24C04读取地址
    I_init();                   // I2C 总线初始化
    if (E_write_block(addr))     // 写入24C04
        printf("write I2C good.\r\n"); // 输出写入成功提示信息
    else

```

```

printf("write I2C bad.\r\n");          // 输出写入失败提示信息
if (E_read_block(addr))                // 读出24C04
printf("read I2C good.\r\n");          // 输出读出成功提示信息
else
printf("read I2C bad.\r\n");            // 输出读出失败提示信息
while(1);
}

```

8.5 I²C接口A/D—D/A转换芯片PCF8591应用编程

PCF8591是具有I²C总线接口的单片、单电源8位A/D—D/A转换器件，具有4路模拟量输入通道、一路模拟量输出通道，3个地址引脚A0、A1和A2用作器件地址，允许将最多8个器件连接至I²C总线，主要特性如下：

- 单电源供电，工作电压为2.5~6V，待机电流低；
- I²C总线接口，采样速率取决于I²C总线速度；
- 4个模拟量输入可编程为单端或差分输入，模拟电压范围为 $V_{SS} \sim V_{DD}$ ；
- 自动增量通道选择；
- 片上跟踪与保持电路；
- 8位逐次逼近式A/D转换器；
- 一路模拟量输出的乘法D/A转换器。

图8.8为PCF8591的引脚排列，各引脚功能见表8.2。

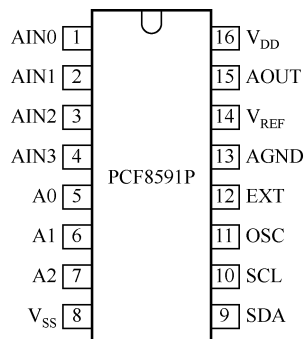


图 8.8 PCF8591 的引脚排列

表8.2 PCF8591各引脚功能

引 脚	功 能
AIN0~AIN3	模拟量输入端
A0、A1、A2	器件地址引脚，可接 V_{DD} 或 V_{SS} ，用于设置从机地址
V_{DD} 、 V_{SS}	正、负电源输入端
AOUT	D/A转换输出端
V_{REF}	参考电压输入端
AGND	模拟地
EXT	片内、外振荡器切换端，接 V_{SS} 时，采用片内振荡器，OSC输出振荡频率；接 V_{DD} 时，OSC切换到高阻态，允许用户连接外部时钟信号
OSC	振荡频率输入/输出端
SCL	I ² C串行时钟
SDA	I ² C串行数据

I²C总线上每一片PCF8591都需要通过发送一个字节的器件地址来激活。I²C总线协议规定器件地址必须在起始条件后作为第一个字节发送。器件地址格式为

1	0	0	1	A2	A1	A0	R/ \overline{W}
---	---	---	---	----	----	----	-------------------

其中，高4位固定为1001，低4位的A0、A1、A2取决于相应引脚所接的电平状态， R/\overline{W}

用于设置数据传输方向， $R/\overline{W}=1$ 为读， $R/\overline{W}=0$ 为写。 I^2C 总线上最多允许接入8个PCF8591芯片。

发送到PCF8591的第二个字节为控制字，用于控制器件功能，格式如图8.9所示。

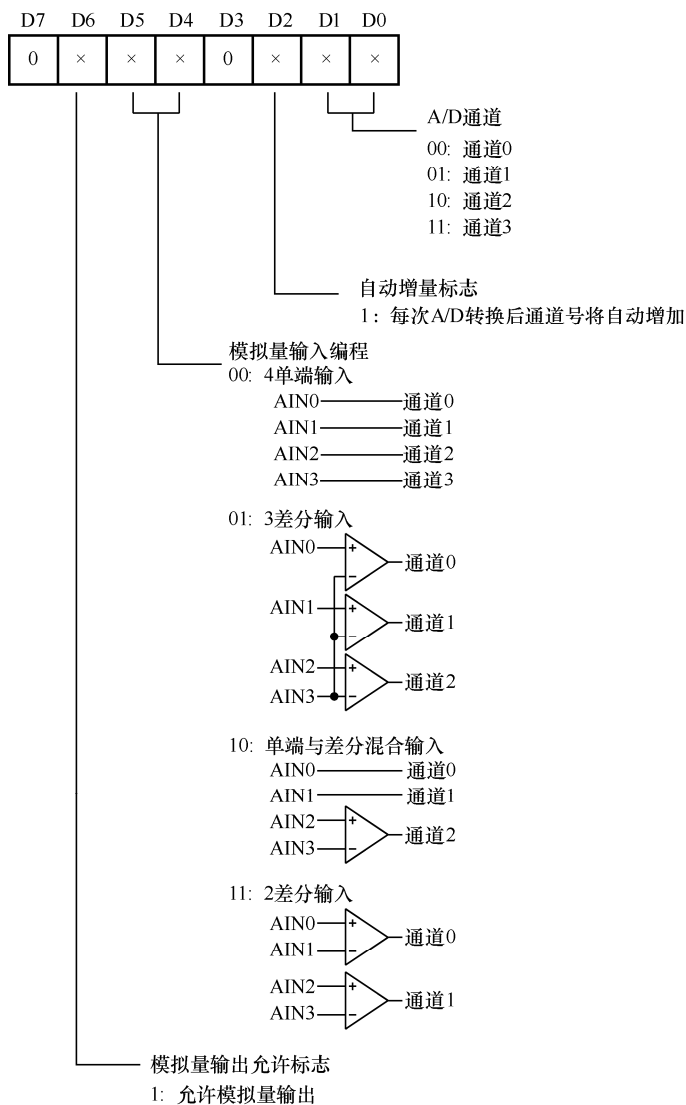


图8.9 PCF8591的控制字格式

控制字高半字节用于设置模拟量输出允许标志、将模拟量输入编程为单端或差分输入；低半字节用于设置自动增量标志、选择模拟输入通道。上电复位后，控制字的初值为0x00。

如果采用PCF8591片内振荡器，并使用了自动增量方式，那么控制字中模拟量输出允许标志应置1，这将使内部振荡器持续运行，此时要防止振荡器启动延时导致的转换错误。模拟量输出允许标志清0，D/A转换器和振荡器将被禁止，模拟量输出为高阻态。

发送到PCF8591的第三个字节作为D/A转换数据被存储到DAC寄存器，并使用片上D/A转换器转换成对应的模拟电压。模拟电压通过AOUT引脚输出，计算公式为

$$V_{\text{AOUT}} = V_{\text{AGND}} + \frac{V_{\text{REF}} - V_{\text{AGND}}}{256} \sum_{i=0}^7 D^i \times 2^i$$

PCF8591的D/A转换时序如图8.10所示。

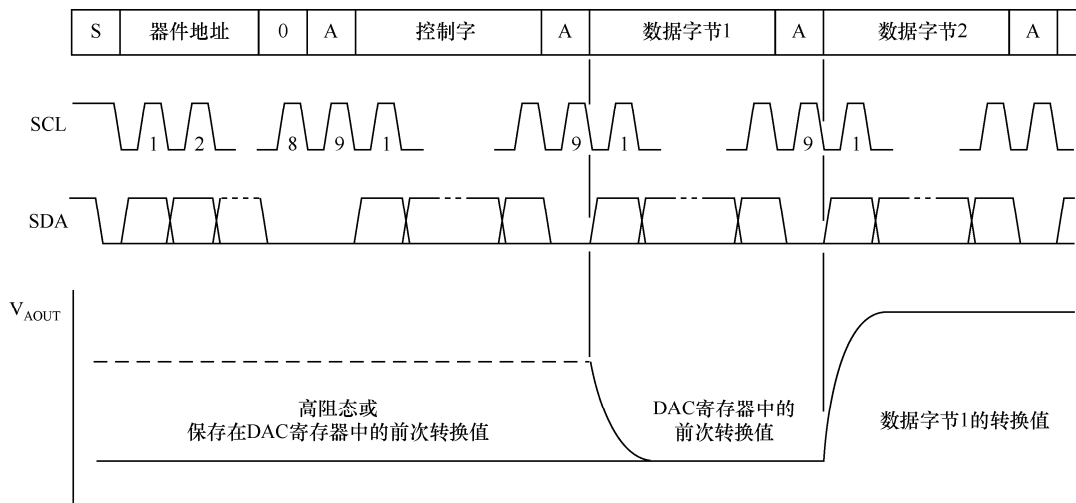


图8.10 PCF8591的D/A转换时序

PCF8591的片内A/D转换器采用逐次逼近转换技术。给PCF8591发送一个有效读方式的器件地址之后，即开始一个A/D转换周期。A/D转换周期在应答时钟脉冲的后沿被触发，并在传送前次转换结果时执行。一旦触发一个转换周期，则所选通道的输入电压将被采样保存到芯片并转换为相应的8位二进制数据。A/D转换数据保存在ADC数据寄存器等待传送，如果自动增量标志置1，将选择下一个通道，在读周期传送的第一个字节包含前一次读周期的转换数据。上电复位之后，读取的第一个字节数据为0x80。A/D转换速率取决于实际的I²C总线速度。

PCF8591的A/D转换时序如图8.11所示。

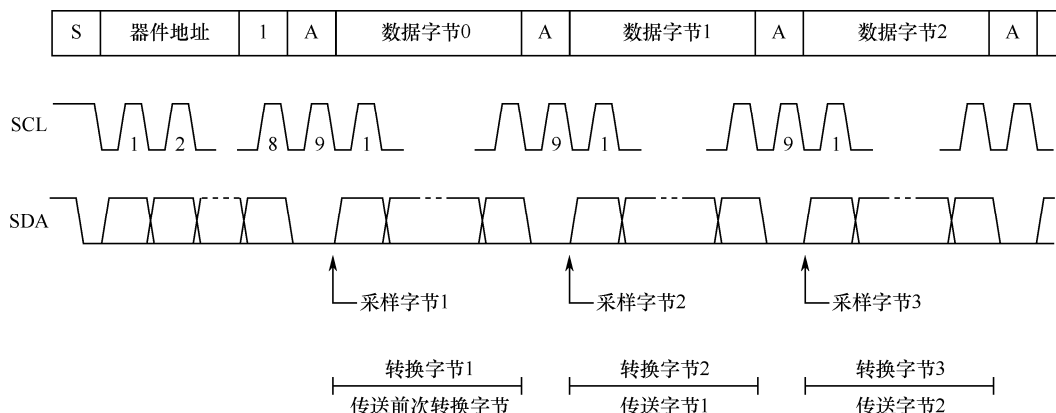


图8.11 PCF8591的A/D转换时序

PCF8591单端输入的A/D转换特性分别如图8.12和图8.13所示。

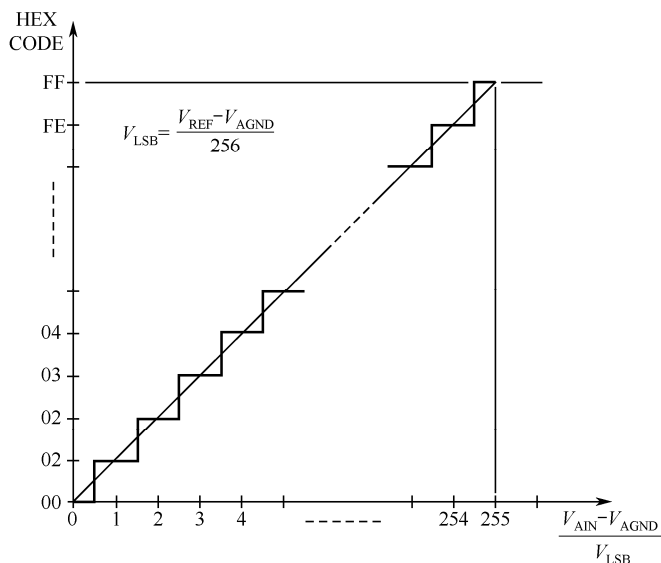


图8.12 PCF8591单端输入的A/D转换特性

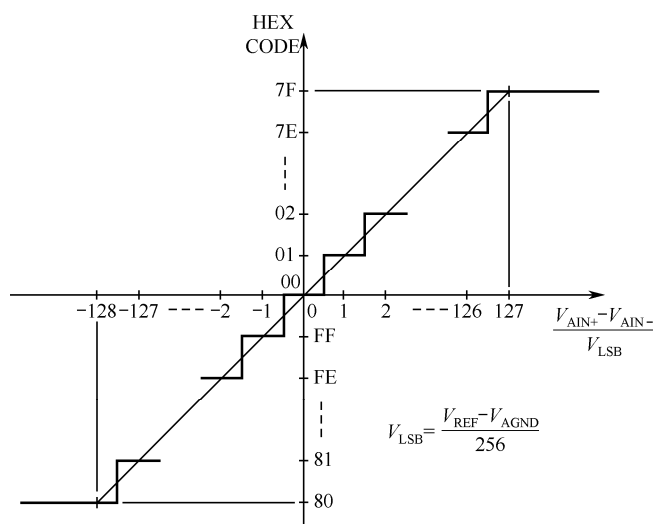
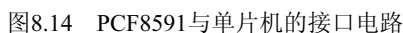


图8.13 PCF8591差分输入的A/D转换特性

图8.14所示为PCF8591与单片机的接口电路。例8-3为对应该电路的C51驱动程序。将PCF8591设置成单端模拟量输入方式，从AIN0~AIN3分别输入4路模拟电压，对应测量值显示在LCD1602液晶屏上，同时从PCF8591的AOUT端输出1路模拟电压。

例8-3 PCF8591的C51驱动程序。

```
#include <reg51.h>
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int
#define delayNOP(); { _nop_(); _nop_(); _nop_(); _nop_(); };
```

267

```

        while (--j);
    } while (--i);
}

/***** Ms延时函数 *****/
void delay(uchar t){
    while(--t) Delaysms();
}

/***** 检查忙状态函数 *****/
bit lcd_busy(){
    bit result;
    LCD_RS = 0;
    LCD_RW = 1;
    LCD_EN = 1;
    delayNOP();
    result = (bit) (P0&0x80);
    LCD_EN = 0;
    return(result);
}

/***** 写指令函数 *****/
void lcd_wcmd(uchar cmd){
    while(lcd_busy());
    LCD_RS = 0;
    LCD_RW = 0;
    LCD_EN = 0;
    _nop_();
    _nop_();
    P0 = cmd;
    delayNOP();
    LCD_EN = 1;
    delayNOP();
    LCD_EN = 0;
}

/***** 写指令函数 *****/
void lcd_wdat(uchar dat){
    while(lcd_busy());
    LCD_RS = 1;
    LCD_RW = 0;
    LCD_EN = 0;
    P0 = dat;
    delayNOP();
    LCD_EN = 1;
    delayNOP();
    LCD_EN = 0;
}

```

```

}

/***** LCD初始化函数 *****/
void lcd_init(){
    delay(15);
    lcd_wcmd(0x38);    //16*2显示, 5*7点阵, 8位数据
    delay(5);
    lcd_wcmd(0x38);
    delay(5);
    lcd_wcmd(0x38);
    delay(5);
    lcd_wcmd(0x0c);    //显示开, 关光标
    delay(5);
    lcd_wcmd(0x06);    //移动光标
    delay(5);
    lcd_wcmd(0x01);    //清除LCD的显示内容
    delay(5);
}

/***** 设定显示位置函数 *****/
void lcd_pos(uchar pos){
    lcd_wcmd(pos | 0x80); //数据指针=80+地址变量
}

/***** 数据处理函数 *****/
void show_value(uchar ad_data){
    dis[2]=ad_data/51;    //AD值转换为3为BCD码, 最大为5.00V。
    dis[2]=dis[2]+0x30;    //转换为ASCII码
    dis[3]=ad_data%51;    //余数暂存
    dis[3]=dis[3]*10;    //计算小数第一位
    dis[1]=dis[3]/51;
    dis[1]=dis[1]+0x30;    //转换为ASCII码
    dis[3]=dis[3]%51;
    dis[3]=dis[3]*10;    //计算小数第二位
    dis[0]=dis[3]/51;
    dis[0]=dis[0]+0x30;    //转换为ASCII码
}

/***** I2C总线启动函数 *****/
void iic_start(void){
    SDA = 1;
    SCL = 1;
    delayNOP();    // 延时
    SDA = 0;
    delayNOP();
    SCL = 0;
}

```

```

/***** I2C总线停止函数 *****/
void iic_stop(void) {
    SDA = 0;
    SCL = 1;
    delayNOP();
    SDA = 1;
    delayNOP();
    SCL = 0;
}

/***** I2C总线初始化函数 *****/
void iicInit(void) {
    SCL = 0;
    iic_stop();
}

/***** I2C总线从机应答函数 *****/
void slave_ACK(void) {
    SDA = 0;
    SCL = 1;
    delayNOP();
    SCL = 0;
}

/***** I2C总线从机非应答函数 *****/
void slave_NOACK(void) {
    SDA = 1;
    SCL = 1;
    delayNOP();
    SDA = 0;
    SCL = 0;
}

/***** I2C总线主机应答位检查函数 *****/
void check_ACK(void) {
    SDA = 1;
    SCL = 1;
    askflag = 0;
    delayNOP();
    if (SDA == 1)    // 若SDA=1表明非应答，置位非应答标志askflag
        askflag = 1;
    SCL = 0;
}

/***** I2C总线发送单字节数据函数 *****/
void IICSendByte(uchar ch) {

```

```

    uchar idata n=8;
    while(n--){
        if((ch&0x80) == 0x80){
            SDA = 1;
            SCL = 1;
            delayNOP();
            SCL = 0;
        }
        else{
            SDA = 0;
            SCL = 1;
            delayNOP();
            SCL = 0;
        }
        ch = ch<<1;
    }
}

/***** I2C总线接收单字节数据函数 *****/
uchar IICreceiveByte(void){
    uchar idata n=8;
    uchar tdata=0;
    while(n--){
        SDA = 1;
        SCL = 1;
        tdata =tdata<<1;
        if(SDA == 1)
            tdata = tdata|0x01;    //若接收到的位为1，则数据的最后一位置1
        else
            tdata = tdata&0xfe;    //否则数据的最后一位置0
        SCL = 0;
    }
    return(tdata);
}

/***** PCF8591发送n位数据函数 *****/
void DAC_PCF8591(uchar controlbyte,uchar w_data){
    iic_start();                //启动I2C
    delayNOP();
    IICSendByte(PCF8591_WRITE); //发送地址位
    check_ACK();                //检查应答位
    if(askflag == 1){
        SystemError = 1;
        return;                 //若非应答，置错误标志位
    }
    IICSendByte(controlbyte&0x77); //发送控制字
    check_ACK();                //检查应答位
}

```

```

        if(askflag == 1){
            SystemError = 1;
            return;                //若非应答, 置错误标志位
        }
        IICSendByte(w_data);      //发送数据
        check_ACK();              //检查应答位
        if(askflag == 1){
            SystemError = 1;
            return;                //若非应答, 置错误标志位
        }
        iic_stop();               //全部发完则停止
        delayNOP();
        delayNOP();
        delayNOP();
        delayNOP();
    }

    /***** PCF8591发送控制字函数 *****/
    void ADC_PCF8591(uchar controlbyte){
        uchar idata receive_da,i=0;
        iic_start();
        IICSendByte(PCF8591_WRITE);    //控制字
        check_ACK();
        if(askflag == 1){
            SystemError = 1;
            return;
        }
        IICSendByte(controlbyte);
        check_ACK();
        if(askflag == 1){
            SystemError = 1;
            return;
        }
        iic_start();                //重新发送开始命令
        IICSendByte(PCF8591_READ);    //控制字
        check_ACK();
        if(askflag == 1){
            SystemError = 1;
            return;
        }
        IICreceiveByte();           //空读一次, 调整读顺序
        slave_ACK();                //收到一个字节后发送一个应答位
        while(i<4){
            receive_da=IICreceiveByte();
            receivebuf[i++]=receive_da;
            slave_ACK();             //收到一个字节后发送一个应答位
        }
    }

```

```

    }
    slave_NOACK();           //收到最后一个字节后发送一个非应答位
    iic_stop();
}

/***** main函数 *****/
main() {
    uchar i, l;
    delay(10);               //延时
    lcd_init();              //初始化LCD
    lcd_pos(0);              //设置显示位置为第一行的第1个字符
    i = 0;
    while(dis4[i] != '\0') { //显示字符
        lcd_wdat(dis4[i]);
        i++;
    }
    lcd_pos(0x40);           //设置显示位置为第二行第1个字符
    i = 0;
    while(dis5[i] != '\0') { //显示字符
        lcd_wdat(dis5[i]);
        i++;
    }
    while(1) {
        ADC_PCF8591(0x44);
        if(SystemError == 1) { //有错误, 重新来
            iicInit();         //I2C总线初始化
            ADC_PCF8591(0x44);
        }
        for(l=0; l<4; l++) {
            show_value(receivebuf[0]); //显示通道0
            lcd_pos(0x02);
            lcd_wdat(dis[2]);          //整数位显示
            lcd_pos(0x04);
            lcd_wdat(dis[1]);          //第一位小数显示
            lcd_pos(0x05);
            lcd_wdat(dis[0]);          //第二位小数显示

            show_value(receivebuf[1]); //显示通道1
            lcd_pos(0x0b);
            lcd_wdat(dis[2]);          //整数位显示
            lcd_pos(0x0d);
            lcd_wdat(dis[1]);          //第一位小数显示
            lcd_pos(0x0e);
            lcd_wdat(dis[0]);          //第二位小数显示
        }
    }
}

```

```

        show_value(receivebuf[2]);    //显示通道2
        lcd_pos(0x42);
        lcd_wdat(dis[2]);            //整数位显示
        lcd_pos(0x44);
        lcd_wdat(dis[1]);            //第一位小数显示
        lcd_pos(0x45);
        lcd_wdat(dis[0]);            //第二位小数显示

        show_value(receivebuf[3]);    //显示通道3
        lcd_pos(0x4b);
        lcd_wdat(dis[2]);            //整数位显示
        lcd_pos(0x4d);
        lcd_wdat(dis[1]);            //第一位小数显示
        lcd_pos(0x4e);
        lcd_wdat(dis[0]);            //第二位小数显示
        iicInit();                    //I2C总线初始化
        if(SystemError == 1){         //有错误,重新来
            iicInit();                //I2C总线初始化
        }
    }
    DAC_PCF8591(0x40, receivebuf[3]); //D/A输出
}
}

```

8.6 I²C接口时钟芯片PCF8563应用编程

PCF8563是一款I²C总线接口、功耗极低的多功能时钟/日历芯片，具有报警、定时、时钟输出、中断输出等多种功能，能完成各种复杂的定时服务。其主要特性如下：

- 宽电压范围为1.0~5.5V，超低功耗；
- 可编程时钟输出频率为32.768kHz、1024Hz、32Hz、1Hz；
- 4种报警和定时器功能；
- 内含复位电路、振荡器电容、掉电检测电路；
- 开漏中断输出；
- 400kHz I²C总线（ $V_{DD}=1.8\sim 5.5V$ ），从地址读为0A3H，从地址写为0A2H。

PCF8563的引脚排列如图8.15所示，各引脚功能见表8.3。

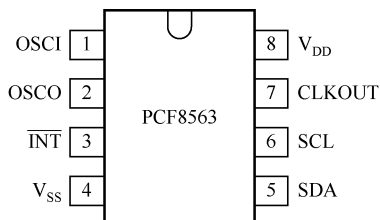


图8.15 PCF8563的引脚排列

表8.3 PCF8563各引脚功能

引 脚	功 能	引 脚	功 能
OSCI	振荡器输入	SCL	I ² C串行时钟
OSCO	振荡器输出	CLKOUT	时钟输出（开漏）
$\overline{\text{INT}}$	中断输出	V _{DD}	正电源
SDA	I ² C串行数据		

PCF8563芯片内部包含有16个8位寄存器、1个可自动增量的地址寄存器、1个内置32.768kHz的振荡器（带有1个内部集成电容）、1个分频器（用于给事实时钟RTC提供源时钟）、1个可变时钟输出、1个定时器、1个报警器、1个掉电检测电路及1个400kHz的I²C总线接口电路。

PCF8563片内16个寄存器占用片内地址00H~0FH单元。其中，00H和01H地址单元为控制/状态寄存器，02H~08H地址单元为秒~年时间寄存器，09H~0CH地址单元为报警功能寄存器，0DH地址单元为时钟输出寄存器，0EH和0FH地址单元为定时器功能寄存器。PCF8563片内各寄存器的功能描述见表8.4。

表8.4 PCF8563片内各寄存器的功能描述

地址	寄 存 器 名	D7	D6	D5	D4	D3	D2	D1	D0
00H	控制/状态寄存器1	TEST1	0	STOP	0	TESTC	0	0	0
01H	控制/状态寄存器2	0	0	0	TI/TP	AF	TF	AIE	TIE
02H	秒寄存器	VL	00~59 （BCD数）						
03H	分寄存器	—	00~59 （BCD数）						
04H	时寄存器	—	—	00~23 （BCD数）					
05H	日寄存器	—	—	01~31 （BCD数）					
06H	星期寄存器	—	—	—	—	—	0~6 （BCD数）		
07H	月寄存器	C	—	—	01~12 （BCD数）				
08H	年寄存器	00~99 （BCD数）							
09H	分报警寄存器	AE	00~59 （BCD数）						
0AH	时报警寄存器	AE	—	00~23 （BCD数）					
0BH	日报警寄存器	AE	—	01~31 （BCD数）					
0CH	星期报警寄存器	AE	—	—	—	—	0~6 （BCD数）		
0DH	时钟输出寄存器	FE	—	—	—	—	—	FD1	FD0
0EH	定时器控制寄存器	TE	—	—	—	—	—	TD1	TD0
0FH	倒计时寄存器	定时器倒计数值（二进制数）							

（1）控制/状态寄存器1

TEST1=0，普通模式，TEST1=1，测试模式；测试模式下，CLKOUT引脚用于输入测试脉冲来取代片内64Hz信号，64个测试脉冲将使秒加1。

STOP=0，芯片时钟正常运行，STOP=1，芯片分频器被异步设置为0，芯片时钟停止运行（CLKOUT在32.768kHz时仍可用）。

TESTC=0, 电源复位功能失效（普通模式下使用），TESTC=1, 电源复位功能有效。

（2）控制/状态寄存器2

TI/TP=0, 当TF有效时, $\overline{\text{INT}}$ 脉冲输出有效（取决于TIE的状态），TI/TP=1, $\overline{\text{INT}}$ 脉冲输出有效（取决于TIE的状态）；若AF和AIE都有效时, 则 $\overline{\text{INT}}$ 脉冲输出一直有效。

AF=1, 表示有报警发生；TF=1, 表示定时器倒数计数结束。AF和TF可用作报警和定时器中断标志, AF和TF状态需要用软件清0。

AIE=0, 报警中断无效；AIE=1, 报警中断有效。

TIE=0, 定时器中断无效；TIE=1, 定时器中断有效。

（3）秒寄存器

VL=0, 保证准确的时钟/日历数据；VL=1, 不保证准确的时钟/日历数据。

（4）月寄存器

C=0, 指定世纪数为20××；C=1, 指定世纪数为19××。

（5）报警寄存器

AE=0, 报警有效；AE=1, 报警无效。

（6）时钟输出寄存器

FE=0, 禁止时钟输出, CLKOUT端被设成高阻抗；FE=1, 允许时钟输出, CLKOUT端输出时钟脉冲。

FD1位和FD0位用于控制CLKOUT端的输出频率, 见表8.5。

表8.5 CLKOUT端输出频率控制

FD1	FD0	CLKOUT端输出频率
0	0	32.768kHz
0	1	1024Hz
1	0	32Hz
1	1	1Hz

（7）倒计时寄存器

TE=0, 定时器无效；TE=1, 定时器有效。

TD1位和TD0位用于定时器时钟频率选择, 见表8.6。

表8.6 定时器时钟频率选择

FD1	FD0	定时器时钟频率
0	0	4096
0	1	64
1	0	1/
1	1	1/60

PCF8563与单片机的接口电路如图8.16所示, 利用单片机P3.0和P3.1引脚模拟I²C总线工作时序, 采用数码管显示RTC时间, 单片机P1用作行列矩阵键盘, h+键和h-键用于小时值加1和减1, m+键和m-键分钟值加1和减1, 支持连续按键。例8-4为对应该电路的C51驱动程序。

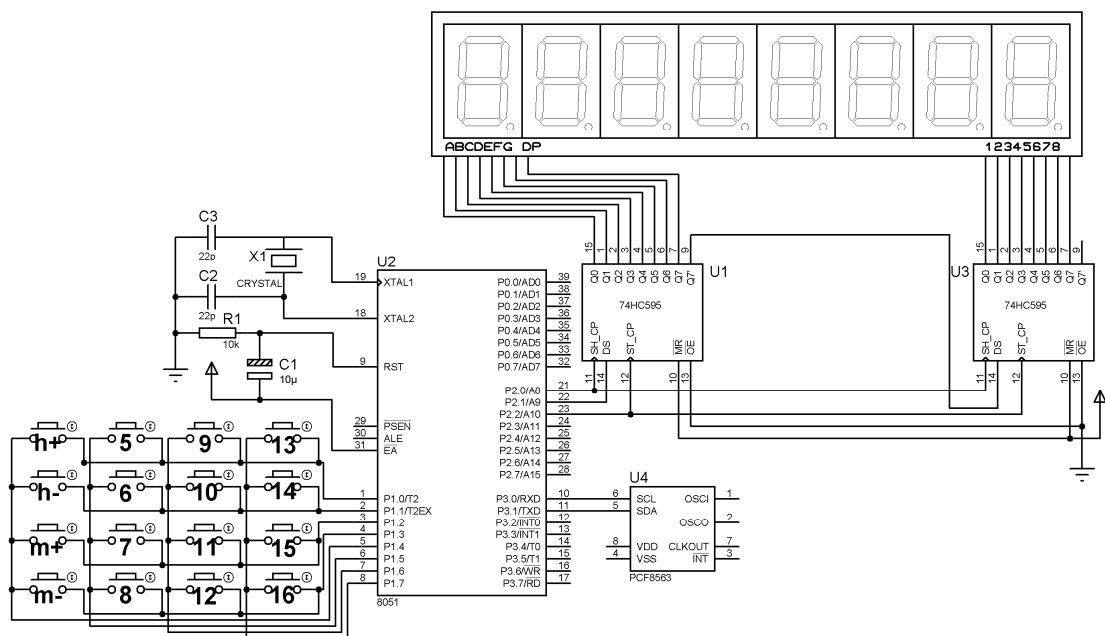


图8.16 PCF8563与单片机的接口电路

例8-4 PCF8563与单片机接口实现RTC的C51驱动程序，包括主模块main.c和RTC时钟模块PCF8563.c。

主模块main.c程序文件如下。

```
#include "reg51.h"
#include "PCF8563.h"
#define LED_TYPE 0x00 //定义LED类型，0x00-共阴，0xff-共阳

/***** 本地常量声明 *****/
uchar code t_display[]={ //标准字库
// 0 1 2 3 4 5 6 7 8 9 A B C D E F
0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x39,0x5E,
0x79,0x71,
//black - H J K L No P U t G Q r M y
0x00,0x40,0x76,0x1E,0x70,0x38,0x37,0x5C,0x73,0x3E,0x78,0x3d,0x67,0x50,
0x37,0x6e,
0xBF,0x86,0xDB,0xCF,0xE6,0xED,0xFD,0x87,0xFF,0xEF,0x46};

//0. 1. 2. 3. 4. 5. 6. 7. 8. 9. -1 //数位码
uchar code T_COM[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};

/***** IO口定义 *****/
sbit DS = P2^1;
sbit ST_CP = P2^2;
sbit SH_CP = P2^0;

/***** 本地变量声明 *****/
```

```

uchar   LED8[8];           //显示缓冲
uchar   display_index;     //显示位索引
bit B_1ms;                 //1ms标志

uchar   KeyState,KeyState1,KeyState2,KeyState3;    //键状态
uchar   KeyHoldCnt;        //键按下计时
uchar   KeyCode;           //给用户使用的键码, 1~16有效
uchar   IO_KeyState, IO_KeyState1, IO_KeyHoldCnt;  //行列键盘变量
uchar   cnt10ms;           //10ms标志
uchar   cnt50ms;           //50ms标志

uchar   hour,minute,second; //RTC变量
uint    msecond;

/*****显示时钟函数*****/
void isplayRTC(void){
    f(hour >= 10)LED8[0] = hour / 10;
    lse          LED8[0] = DIS_BLACK;
    ED8[1] = hour % 10;
    ED8[2] = DIS_;
    ED8[3] = minute / 10;
    ED8[4] = minute % 10;
    ED8[6] = second / 10;
    ED8[7] = second % 10;
}

/*****读RTC函数*****/
void eadRTC(void){
    char   tmp[3];
    eadNbyte(2, tmp, 3);
    econd = ((tmp[0] >> 4) & 0x07) * 10 + (tmp[0] & 0x0f);
    inute = ((tmp[1] >> 4) & 0x07) * 10 + (tmp[1] & 0x0f);
    our   = ((tmp[2] >> 4) & 0x03) * 10 + (tmp[2] & 0x0f);
}

/*****写RTC函数*****/
void riteRTC(void){
    char   tmp[3];
    mp[0] = ((second / 10) << 4) + (second % 10);
    mp[1] = ((minute / 10) << 4) + (minute % 10);
    mp[2] = ((hour / 10) << 4) + (hour % 10);
    riteNbyte(2, tmp, 3);
}

uchar code T_KeyTable[16] = {0,1,2,0,3,0,0,0,4,0,0,0,0,0,0,0};

/*****键扫描延时函数*****/
void IO_KeyDelay(void){
    char i;
    = 60;

```

```

    hile(--i) ;
}

/*****键扫描函数*****/
void IO_KeyScan(void) {
    uchar    j;
    j = IO_KeyStat1;           //保存上一次状态
    P1 = 0xf0;
    IO_KeyDelay();
    IO_KeyStat1 = P1 & 0xf0;
    P1 = 0x0f;
    IO_KeyDelay();
    IO_KeyStat1 |= (P1 & 0x0f);
    IO_KeyStat1 ^= 0xff;       //取反

    if(j == IO_KeyStat1){     //连续两次读相等
        j = IO_KeyState;
        IO_KeyState = IO_KeyStat1;
        if(IO_KeyState != 0){ //有键按下
            F0 = 0;
            if(j == 0) F0 = 1; //第一次按下
            else if(j == IO_KeyState){
                if(++IO_KeyHoldCnt >= 20) { //1秒后重键
                    IO_KeyHoldCnt = 18;
                    F0 = 1;
                }
            }
            if(F0){
                j = T_KeyTable[IO_KeyState >> 4];
                if((j != 0) && (T_KeyTable[IO_KeyState & 0x0f] != 0))
                    KeyCode = (j-1)*4+T_KeyTable[IO_KeyState & 0x0f];
            }
            else IO_KeyHoldCnt = 0;
        }
        P1 = 0xff;
    }
}

/*****向HC595发送一个字节*****/
void Send_595(uchar dat){
    uchar    i;
    for(i=0; i<8; i++){
        dat <<= 1;
        DS    = CY;
        SH_CP = 1;
        SH_CP = 0;
    }
}

/*****显示扫描函数*****/

```

```

void DisplayScan(void){
    Send_595(~LED_TYPE ^ T_COM[display_index]);          //输出位码
    Send_595( LED_TYPE ^ t_display[LED8[display_index]]); //输出段码
    ST_CP = 1;
    ST_CP = 0;                                           //锁存输出数据
    if(++display_index >= 8)    display_index = 0; //8位结束, 回0
}

/***** T0 1ms中断函数 *****/
void timer0 (void) interrupt 1 {
    TH0 = 0xfc;          //重装1ms初值
    TL0 = 0x18;
    DisplayScan();       //扫描显示一位
    B_1ms = 1;           //1ms标志
}

/*****主函数*****/
void main(void){
    uchar    i;
    display_index = 0;
    TMOD = 0x01;          //T0工作于方式1
    TH0 = 0xfc;           //定时1ms初值
    TL0 = 0x18;
    ET0 = 1;
    TR0 = 1;
    EA = 1;               //开中断
    for(i=0; i<8; i++) LED8[i] = 0x10;
    ReadRTC();
    F0 = 0;
    if(second >= 60)    F0 = 1; //错误
    if(minute >= 60)    F0 = 1; //错误
    if(hour >= 60) F0 = 1;      //错误
    if(F0){              //有错误, 默认12:00:00
        second = 0;
        minute = 0;
        hour = 12;
        WriteRTC();
    }
    DisplayRTC();
    LED8[2] = DIS_;
    LED8[5] = DIS_;
    KeyState = 0;
    KeyState1 = 0;
    KeyState2 = 0;
    KeyState3 = 0;        //键状态
    KeyHoldCnt = 0;       //键按下计时
    KeyCode = 0;          //给用户使用的键码, 1~16有效
    cnt10ms = 0;
    IO_KeyState = 0;
    IO_KeyState1 = 0;

```

```

IO_KeyHoldCnt = 0;
cnt50ms = 0;
while(1){
    if(B_1ms){ //1ms到
        B_1ms = 0;
        if(++msecond >= 1000){ //1秒到
            msecond = 0;
            ReadRTC();
            DisplayRTC();
        }
        if(++cnt50ms >= 50){ //50ms扫描一次行列键盘
            cnt50ms = 0;
            IO_KeyScan();
        }
        if(KeyCode > 0){ //有键按下

            LED8[6] = KeyCode / 10; //显示键码
            LED8[7] = KeyCode % 10; //显示键码

            if((KeyCode == 1)){ //时+1
                if(++hour >= 24) hour = 0;
                WriteRTC();
                DisplayRTC();
            }
            if((KeyCode == 2)){ //时-1
                if(--hour >= 24) hour = 23;
                WriteRTC();
                DisplayRTC();
            }
            if((KeyCode == 3)){ //分+1
                second = 0;
                if(++minute >= 60) minute = 0;
                WriteRTC();
                DisplayRTC();
            }
            if((KeyCode == 4)){ //分-1
                second = 0;
                if(--minute >= 60) minute = 59;
                WriteRTC();
                DisplayRTC();
            }
            KeyCode = 0;
        }
    }
}

```

RTC时钟模块PCF8563.c程序文件如下。

```

#include "PCF8563.h"
#include "STC15F2K.h"

```

```

#define MAIN_Fosc      22118400L    //定义主时钟

sbit    SDA = P3^1;    //定义SDA
sbit    SCL = P3^0;    //定义SCL

/*****I2C总线延时函数*****/
void    I2C_Delay(void) {
    uchar    dly;
    dly = MAIN_Fosc / 2000000UL;      //延时2us
    while(--dly)    ;
}

/*****I2C总线启动函数*****/
void I2C_Start(void) {
    SDA = 1;
    I2C_Delay();
    SCL = 1;
    I2C_Delay();
    SDA = 0;
    I2C_Delay();
    SCL = 0;
    I2C_Delay();
}

/*****I2C总线停止函数*****/
void I2C_Stop(void) {
    SDA = 0;
    I2C_Delay();
    SCL = 1;
    I2C_Delay();
    SDA = 1;
    I2C_Delay();
}

/*****I2C总线应答函数*****/
void S_ACK(void) {
    SDA = 0;
    I2C_Delay();
    SCL = 1;
    I2C_Delay();
    SCL = 0;
    I2C_Delay();
}

/*****I2C总线非应答函数*****/
void S_NoACK(void) {
    SDA = 1;
    I2C_Delay();
    SCL = 1;

```



```

    I2C_Delay();
    SCL = 0;
    I2C_Delay();
}

/***** I2C总线应答检查函数 *****/
void I2C_Check_ACK(void){
    SDA = 1;
    I2C_Delay();
    SCL = 1;
    I2C_Delay();
    F0 = SDA;
    SCL = 0;
    I2C_Delay();
}

/***** I2C总线单字节写函数 *****/
void I2C_WriteAbyte(uchar dat){
    uchar i;
    i = 8;
    do
    {
        if(dat & 0x80) SDA = 1;
        else          SDA = 0;
        dat <<= 1;
        I2C_Delay();
        SCL = 1;
        I2C_Delay();
        SCL = 0;
        I2C_Delay();
    }
    while(--i);
}

/***** I2C总线单字节读函数 *****/
uchar I2C_ReadAbyte(void){
    uchar i,dat;
    i = 8;
    SDA = 1;
    do
    {
        SCL = 1;
        I2C_Delay();
        dat <<= 1;
        if(SDA)    dat++;
        SCL = 0;
        I2C_Delay();
    }
    while(--i);
    return(dat);
}

```

```

    }

    /***** I2C总线N字节写函数 *****/
    void WriteNbyte(uchar addr, uchar *p, uchar number){
        I2C_Start();
        I2C_WriteAbyte(SLAW);
        I2C_Check_ACK();
        if(!F0){
            I2C_WriteAbyte(addr);
            I2C_Check_ACK();
            if(!F0){
                do{
                    I2C_WriteAbyte(*p);
                    p++;
                    I2C_Check_ACK();
                    if(F0) break;
                }
                while(--number);
            }
        }
        I2C_Stop();
    }

    /***** I2C总线N字节读函数 *****/
    void ReadNbyte(uchar addr, uchar *p, uchar number){
        I2C_Start();
        I2C_WriteAbyte(SLAW);
        I2C_Check_ACK();
        if(!F0){
            I2C_WriteAbyte(addr);
            I2C_Check_ACK();
            if(!F0){
                I2C_Start();
                I2C_WriteAbyte(SLAR);
                I2C_Check_ACK();
                if(!F0){
                    do{
                        *p = I2C_ReadAbyte();
                        p++;
                        if(number != 1) S_ACK(); //发送应答
                    }
                    while(--number);
                    S_NoACK(); //发送非应答
                }
            }
        }
        I2C_Stop();
    }

```

Proteus仿真设计实例

单片机应用系统设计会同时涉及硬件和软件技术，利用Proteus虚拟仿真技术，可以在没有单片机实际硬件的条件下，利用个人计算机实现单片机软件和硬件同步仿真，仿真结果可以直接应用于真实设计，极大地提高了单片机应用系统的设计效率。

9.1 红外遥控系统

9.1.1 功能要求

设计一套NEC格式的红外遥控系统，要求以8051单片机作为遥控发射和接收的主控制器，利用单片机内部定时器功能实现发射编码和接收解码，通过键盘按键启动发射，通过LCD和数码管分别显示发射和接收到的数据。

9.1.2 硬件电路设计

红外遥控系统由遥控电路和接收电路组成。红外遥控电路如图9.1所示，包括8051单片机、DSW拨码开关、LCD显示器、矩阵键盘、红外发射管等。单片机通过编程实现NEC格式的遥控编码，拨码开关DSW1和DSW2用于设定16位用户码。工作时，矩阵键盘每个按键产生不同的操作码，通过红外发射管发射出去，同时连同16位用户码在LCD显示器上显示出来。

红外接收电路如图9.2所示，包括8051单片机、数码管显示器、LED指示灯等。单片机接收到NEC格式的红外线脉冲信号后，通过编程实现解码，并将解码数值显示在数码管显示器上，同时通过LED指示灯显示遥控电路中的C键是长按还是短按。

9.1.3 软件程序设计

红外遥控系统程序设计包括编码程序和解码程序。编码程序按规定的格式，为键盘中每个按键设置相应的码值并转换成红外线脉冲信号发射出去；解码程序则根据接收到的红外线脉冲还原键码，并根据不同的键码实现不同的按键功能。

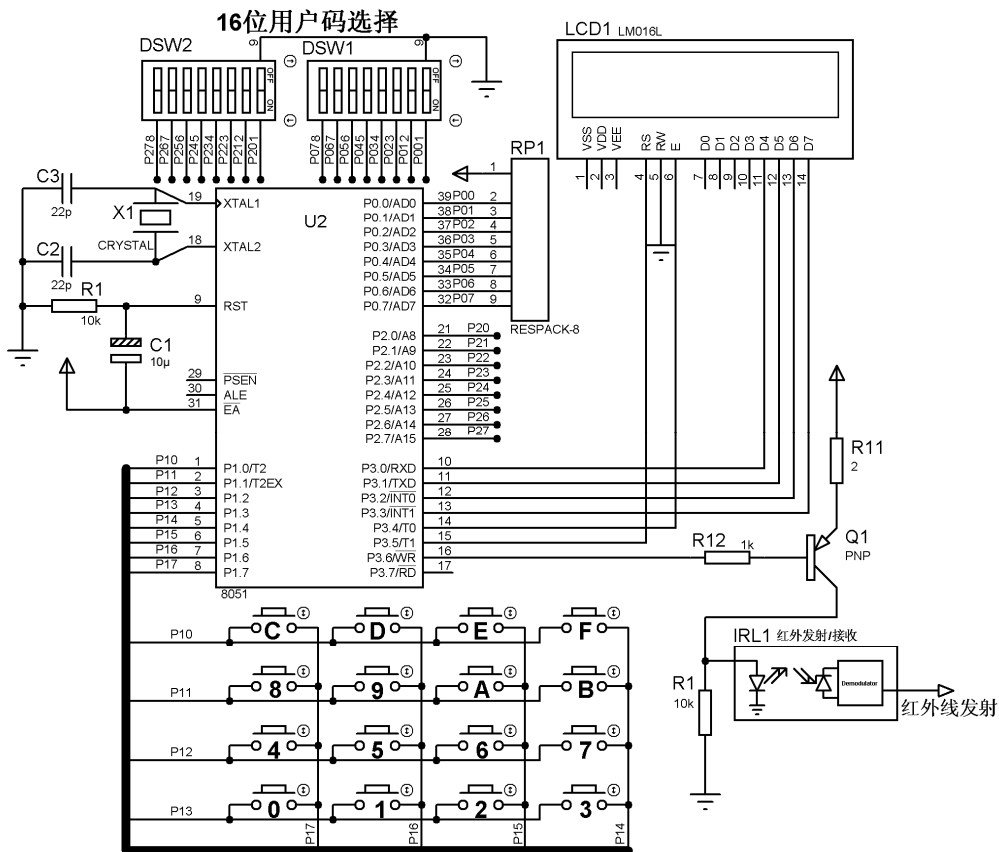


图9.1 红外遥控电路

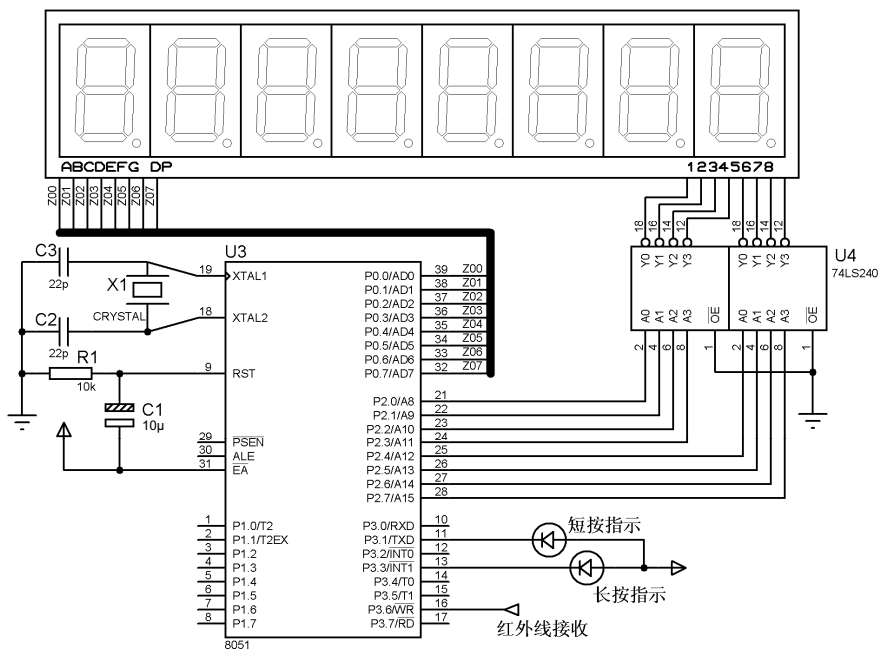


图9.2 红外接收电路

红外遥控系统的数据格式如图9.3所示，包括起始引导码、用户码、数据码和数据码反码。引导码为9ms高电平加4.5ms低电平，用户码为16位，数据码和数据反码各为8位。数据反码主要用于判断接收的数据是否正确。

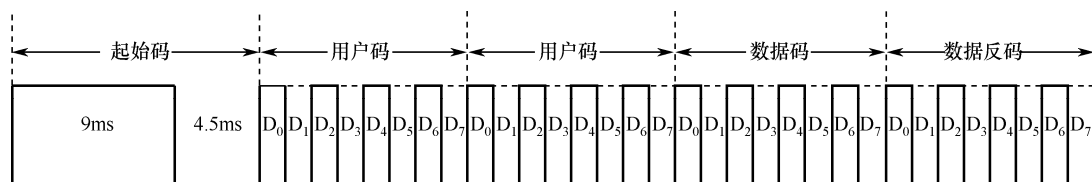


图9.3 红外遥控系统的数据格式

用户码或数据码中的每一位可以是1，也可以是0。位0用0.56ms高电平加0.56ms低电平表示；位1用0.56ms高电平加1.68ms低电平表示，如图9.4所示。

根据以上数据格式和位电平，就可以采用C51编写出遥控系统应用程序，包括遥控发射和遥控解码程序。在遥控发射程序中，预先对键盘各个按键设置遥控操作的数据码，再根据图9.3的数据格式加入起始码和用户码，并根据如图9.4所示数据“1”和“0”的高、低电平格式，生成一系列串行脉冲，通过遥控电路单片机的引脚P3.6不断向外发送，同时将发送出去的代码通过LCD显示出来。

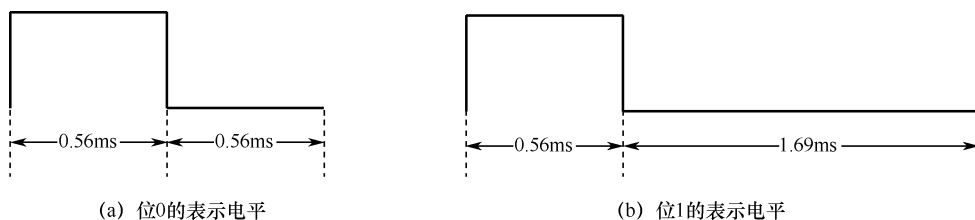


图9.4 数据格式中位0和位1的表示电平

在遥控解码程序中，由遥控接收电路单片机的引脚P3.6不断接收脉冲信号，并根据如图9.4所示格式将脉冲信号还原为二进制代码，通过LED数码管显示出来，从解码得到的数据中截取所需要的遥控操作数据码，即可完成不同的操作功能。

遥控编码数据采用延时方式完成，而遥控解码数据则采用定时中断方式完成。

例9-1 红外遥控系统的软件程序。

(1) 遥控发射的C51源程序文件清单。

```
#include <reg51.h>
#include <intrins.h>

#define uint8    unsigned char
#define uint8c   unsigned char code
#define uint16   unsigned int
#define uint32   unsigned long
#define USER_H  P2           //用户码高8位
#define USER_L  P0           //用户码低8位
#define NOP      _nop_()
#define m9       (65536-9000) //9ms
```

```

#define m4_5 (65536-4500) //4.5ms
#define m1_6 (65536-1650) //1.65ms
#define m_56 (65536-560) //0.56ms
#define m40 (65536-40000) //40ms
#define m56 (65536-56000) //56ms
#define m2_25 (65536-2250) //2.25ms
#define L1602_IO P3 //LCD使用低4位IO口

uint8c asc[16]={ "0123456789ABCDEF" };
uint8c tab[16] = { //遥控操作数据码
    0x12,0x05,0x1e,0x55,
    0x01,0x1b,0x03,0x6b,
    0x07,0x08,0x09,0x68,
    0x22,0xE6,0x33,0xe2
};

/*****1ms延时函数*****/
void YS1ms(uint8 n){
    uint8 i,j,k;
    for(i=n;i!=0;i--)
        for(j=12;j!=0;j--)
            for(k=248;k!=0;k--);
}

/*****LCD 4位数据写入函数*****/
void L1602_WR4bits(uint8 dat){
    L1602_E = 1;
    L1602_IO &= 0xF0; //清IO端口低4位上的数据
    dat &= 0x0F;
    L1602_IO |= dat; //给IO端口低4位送数据
    NOP;
    L1602_E = 0; //下降沿有效
    YS1ms(1);
}

/*****LCD 指令写入函数*****/
void L1602_cmd(uint8 cmd){
    L1602_RS = 0; //允许写指令
    L1602_WR4bits(cmd>>4); //传指令高4位
    L1602_WR4bits(cmd); //传指令低4位
    L1602_RS = 1; //允许写数据
}

/*****LCD清屏函数*****/
void L1602_clr(void){
    L1602_cmd( 0x01); //清屏指令: 0x01
}

```

```

/*****LCD显示位置函数*****/
void L1602_xy(uint8 x,bit y){ //y=1: 第二行, y=0: 第一行
    if(y)L1602_cmd(x|0xC0); //数据指针 = 0xC0+地址码
    else L1602_cmd(x|0x80); //数据指针 = 0x80+地址码(00H~27H, 40H~67H)
}

/*****LCD字符输出函数*****/
void L1602_ZIFU(uint8 dat){
    L1602_WR4bits(dat>>4);
    L1602_WR4bits(dat);
}

/*****LCD字符串输出函数*****/
void L1602_ZIFUC(uint8 *s){
    while(*s){
        L1602_WR4bits(*s>>4);
        L1602_WR4bits(*s);
        s++;
    }
}

/*****LCD整型数据显示函数*****/
void L1602_JZ(uint32 tem,uint8 num,uint8 i){
    uint8 j,k,z[10];
    bit BT=0;
    switch(num){
        case 2: tem = _lror_(tem,i-1);
            while(i--){
                if(tem & 1)L1602_ZIFU( '1' );
                else L1602_ZIFU( '0' );
                tem = _lrol_(tem,1);
            }
            break;
        case 10: for(j=0;j<i;j++){
            z[j] = tem%10; //z[0]=个位, z[1]=十位, .....
            tem /= 10;
        }
        while(i--){
            if(z[i] || (i==0))BT=1; //数据有效标志
            if(BT)L1602_ZIFU(asc[z[i]]); //数据有效前的“0”不显示
            else L1602_ZIFU( ' ' ); //用空格替换。
        }
        break;
        case 16: for(j=4-i;j<4;j++){
            k = ((uint8 *)&tem)[j];
            L1602_ZIFU(asc[k/16]); //发送高4位
        }
    }
}

```

```

        L1602_ZIFU(asc[k%16]);          //发送低4位
    }
    break;
}
}

/*****LCD初始化函数*****/
void L1602_Init(void) {
    YS1ms(15);                          //延时15ms
    L1602_RS = 0;
    YS1ms(1);
    L1602_WR4bits( 0x30>>4);
    YS1ms(5);
    L1602_WR4bits( 0x30>>4);
    YS1ms(5);
    L1602_WR4bits( 0x30>>4);
    YS1ms(5);
    L1602_WR4bits( 0x20>>4);          //设置LCD为4位数据总线方式
    YS1ms(5);
    L1602_cmd( 0x28);                  //设置显示 2行, 5*7点阵
    L1602_cmd( 0x0C);
    L1602_cmd( 0x06);
    L1602_RS = 1;
    YS1ms(1);
    L1602_clr();                        //清屏
}

/*****38KHz载波发射函数*****/
void TT0(bit BT,uint16 x){
    TH0 = x>>8;                        //输入T0初始值
    TL0 = x;
    TF0=0;                              //清0
    TR0=1;                              //启动定时器0
    if(BT == 0) while(!TF0); //BT=0时, 不发射38KHz载波; BT=1发射38KHz载波
                                且延时
    else while(1){                    //38KHz载波, (低电平) 占空比5:26
        IR = 0;
        if(TF0)break;if(TF0)break;
        IR = 1;
        if(TF0)break;if(TF0)break;
        if(TF0)break;if(TF0)break;
        if(TF0)break;if(TF0)break;
        if(TF0)break;if(TF0)break;
        if(TF0)break;if(TF0)break;
        if(TF0)break;if(TF0)break;
    }
    TR0=0;                              //关闭定时器0
    TF0=0;                              //标志位溢出则清0
}

```



```

    IR = 1;                //载波停止后, 发射端口常态为高
}

/*****8位数据发送程序函数*****/
void Z0(uint8 temp){
    uint8 v;
    for (v=0;v<8;v++){
        TT0(1,m_56);        //循环8次移位
        if(temp&0x01) TT0(0,m1_6);    //高电平0.65mS
        else TT0(0,m_56);    //发送最低位
        temp >>= 1;        //右移一位
    }
}

/*****线翻转法按键识别函数*****/
uint8 KEY(void){
    uint8 Key = 0;
    P1 = 0xf0;            //键盘初始: 行值=0, 列值=1
    NOP;                  //缓冲, 待IO端口电位稳定
    Key = P1&0xf0;        //得到行标志
    P1 = 0x0f;            //翻转键盘接口输出
    NOP;
    Key |= (P1&0x0f);     //列标志 + 行标志
    return Key;           //返回键值
}

/*****NEC格式遥控编码发送函数*****/
void ZZ(uint8 Value){
    L1602_xy(12,1);
    L1602_JZ(Value,16,1); //更新显示
    TT0(1,m9);            //高电平9mS
    TT0(0,m4_5);         //低电平4.5mS
    Z0(USER_H);           //用户码高8位
    Z0(USER_L);           //用户码低8位
    Z0(Value);            //数据码
    Z0(~Value);           //数据码反码
    TT0(1,m_56);         //结束码
    TT0(0,m40);
    while(KEY() != 0xFF){ //重复码
        TT0(1,m9);
        TT0(0,m2_25);
        TT0(1,m_56);
        TT0(0,m40);
        TT0(0,m56);
    }
}

```

```

/*****键值散转函数*****/
void SanZhuan() {
    uint8 v;
    v = KEY();          //键盘检测
    switch(v) {
        case 0x7e:ZZ(tab[0]);break;
        case 0xbe:ZZ(tab[1]);break;
        case 0xde:ZZ(tab[2]);break;
        case 0xee:ZZ(tab[3]);break;
        case 0x7d:ZZ(tab[4]);break;
        case 0xbd:ZZ(tab[5]);break;
        case 0xdd:ZZ(tab[6]);break;
        case 0xed:ZZ(tab[7]);break;
        case 0x7b:ZZ(tab[8]);break;
        case 0xbb:ZZ(tab[9]);break;
        case 0xdb:ZZ(tab[10]);break;
        case 0xeb:ZZ(tab[11]);break;
        case 0x77:ZZ(tab[12]);break;
        case 0xb7:ZZ(tab[13]);break;
        case 0xd7:ZZ(tab[14]);break;
        case 0xe7:ZZ(tab[15]);break;
        default:break;
    }
    v=0;
}

/*****主函数*****/
void main(void) {
    TMOD = 0x01;          //T0 16位工作方式
    IR=1;                 //发射端口常态为高电平
    L1602_Init();
    L1602_clr();
    L1602_xy(0,0);
    L1602_ZIFUC("UserCode :0x");
    L1602_xy(0,1);
    L1602_ZIFUC("Opcode :0x");
    while(1) {
        L1602_xy(12,0);
        L1602_JZ(USER_H,16,1);
        L1602_JZ(USER_L,16,1);
        SanZhuan();
    }
}

```

(2) 遥控解码C51源程序文件清单。

```

#include "reg51.h"
#include <intrins.h>

```

```

sfr SE = 0x80;          //数码管段选 P0:0x80 P1:0x90
sbit WX1 = P2^0;        //数码管位显
sbit WX2 = P2^1;
sbit WX3 = P2^2;
sbit WX4 = P2^3;
sbit WX5 = P2^4;
sbit WX6 = P2^5;
sbit WX7 = P2^6;
sbit WX8 = P2^7;
sbit P33 = P3^3;
sbit P31 = P3^1;

#define USER_H 0x80      //用户码高8位
#define USER_L 0x7F      //用户码低8位
#define Check_EN 0        //是否要校验16位用户码: 不校验填0, 校验则填1
#define CPU_Fosc 12000000 //输入主频, 范围6MHz~40MHz
#define CA_S 4            //长按时间设置, 108ms的整数倍, 10倍以上为宜
#define Step 400          //红外采样步长, 400us
#define TH_H ((65536-Step*(CPU_Fosc/300)/40000)/256)
//定时器高8位基准初值
#define TH_L ((65536-Step*(CPU_Fosc/300)/40000)%256)
//定时器低8位基准初值
#define Boot_Limit (((9000+4500)+2000)/Step) //引导码周期上限
#define Boot_Lower (((9000+4500)-2000)/Step) //引导码周期下限
#define Bit1_Limit ((2250+800)/Step) //“1”周期上限
#define Bit0_Limit ((1125+400)/Step) //“0”周期上限
#define NOP _nop_()

#define uint8 unsigned char
#define uint8c unsigned char code
#define uint16 unsigned int

uint8 IR_BT;          //解码效果返回值: 0无效, 1有效, 2短按, 3长按
uint8 NEC[4];         //解码存放数组, 16位用户码+操作码正反码
uint8 cntCA;          //长按计数
uint16 cntStep;       //步数
bit IRa, IRb;         //电平状态
bit IRsync;           //同步标志
uint8 BitN;           //位码装载数
uint8c tab[] = {0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8,
                0x80, 0x90, 0x88, 0x83, 0xc6, 0xa1, 0x86, 0x8e, 0xff};
uint8 Xn, X1, X2, X3, X4, X5, X6;

/*****定时器初始化函数*****/
void IR_Init() {
    TMOD = 0x01;      //设置T0为16位定时器方式

```

```

    TL0 = TH_L;      //每步时间
    TH0 = TH_H;
    ET0 = 1;        //开中断
    EA = 1;
    TR0 = 1;
}

/*****红外解码函数*****/
void IR_NEC() {
    TL0 = TH_L;      //T0重赋初值
    TH0 = TH_H;
    cntStep++;       //步数采样
    if (IR_BT==1) if (cntStep>300) IR_BT=2; //如果无长按, 120ms后默认为短按
    IRb = IRa;       //上次电平状态
    IRa = IR;        //当前电平状态
    if (IRb && !IRa) { //是否下降沿 (上次高, 当前低)
        if (cntStep > Boot_Limit) { //超过同步时间?
            if (IR_BT==1) if (++cntCA>CA_S) IR_BT=3; //长按
            IRsync=0; //同步位清0
        }
        else if (cntStep > Boot_Lower) { IRsync=1; BitN=32; } //同步位置1
        else if (IRsync) { //如果已同步
            if (cntStep > Bit1_Limit) IRsync=0;
            else {
                NEC[3] >>= 1;
                if (cntStep > Bit0_Limit) NEC[3] |= 0x80; //“0”与“1”
                if (--BitN == 0) {
                    IRsync = 0; //同步位清0
                }
            }
        }
        #if (Check_EN == 1)
            if ((NEC[0]==USER_H) && (NEC[1]==USER_L) && (NEC[2]==~NEC[3]))
                //校验用户码、操作码
                { IR_BT=1; cntCA=0; } //解码有效, 判断短按? 长按?
        #else
            if (NEC[2]==~NEC[3]) { IR_BT=1; cntCA=0; } //校验数据码正反码
        #endif
    }
    else if ((BitN & 0x07) == 0) //NEC[3]每装满8位移动保存一次
    { NEC[0]=NEC[1]; NEC[1]=NEC[2]; NEC[2]=NEC[3]; }
}
cntStep = 0; //步数计清0
}

/*****遥控短按处理函数*****/
void KZ0() {
    switch (NEC[2]) {

```

```

        case 0x12: P31 = !P31; break;
        default: break;
    }
}

/*****遥控长按处理函数*****/
void KZ1() {
    switch(NEC[2]) {
        case 0x12: P33 = !P33; break;
        default: break;
    }
}

/*****数码管扫描函数*****/
void XS(void) {
    if(++Xn > 7) Xn=0;
    switch(Xn) {
        case 0: WX8=1; NOP;           //屏蔽上个位显
                SE=tab[X1];           //送段码
                WX1=0;                 //开位显
                break;
        case 1: WX1=1; NOP; SE=tab[X2]; WX2=0; break;
        case 2: WX2=1; NOP; SE=tab[X3]; WX3=0; break;
        case 3: WX3=1; NOP; SE=tab[X4]; WX4=0; break;
        case 4: WX4=1; NOP; SE=tab[X5]; WX5=0; break;
        case 5: WX5=1; NOP; SE=tab[X6]; WX6=0; break;
        case 6: WX6=1; NOP; SE=tab[X7]; WX7=0; break;
        case 7: WX7=1; NOP; SE=tab[X8]; WX8=0; break;
        default: break;
    }
}

/*****定时器T0中断函数*****/
void time0(void) interrupt 1 {
    IR_NEC();
    XS();
}

/*****主函数*****/
void main(void) {
    IR_Init();           //红外线解码初始化
    while(1) {           //遥控检测
        if((IR_BT==2) || (IR_BT==3)) {
            if(IR_BT==2) KZ0(); //短按处理
            else KZ1();         //长按处理
            IR_BT = 0;          //清有效标志
            X1 = NEC[0]/16;     //更新显示
        }
    }
}

```

```

        X2 = NEC[0]%16;
        X3 = NEC[1]/16;
        X4 = NEC[1]%16;
        X5 = NEC[2]/16;
        X6 = NEC[2]%16;

    }

}

}

```

9.2 点阵LED显示屏

9.2.1 功能要求

利用8051单片机片内定时器和IO端口，设计一个16×16点阵LED显示屏，要求能够稳定地显示图形和文字。显示方式可为静态显示，也可以向上滚动显示。

9.2.2 硬件电路设计

图9.5为16×16点阵LED显示屏的硬件电路图。8051单片机是整个电路的核心，1片74HC154与P2口相连，用作LED显示屏的行驱动。74HC154是4/16译码器。其16根译码输出线用来选通LED显示屏的16行。两片74HC595与单片机P3口相连，用作LED显示

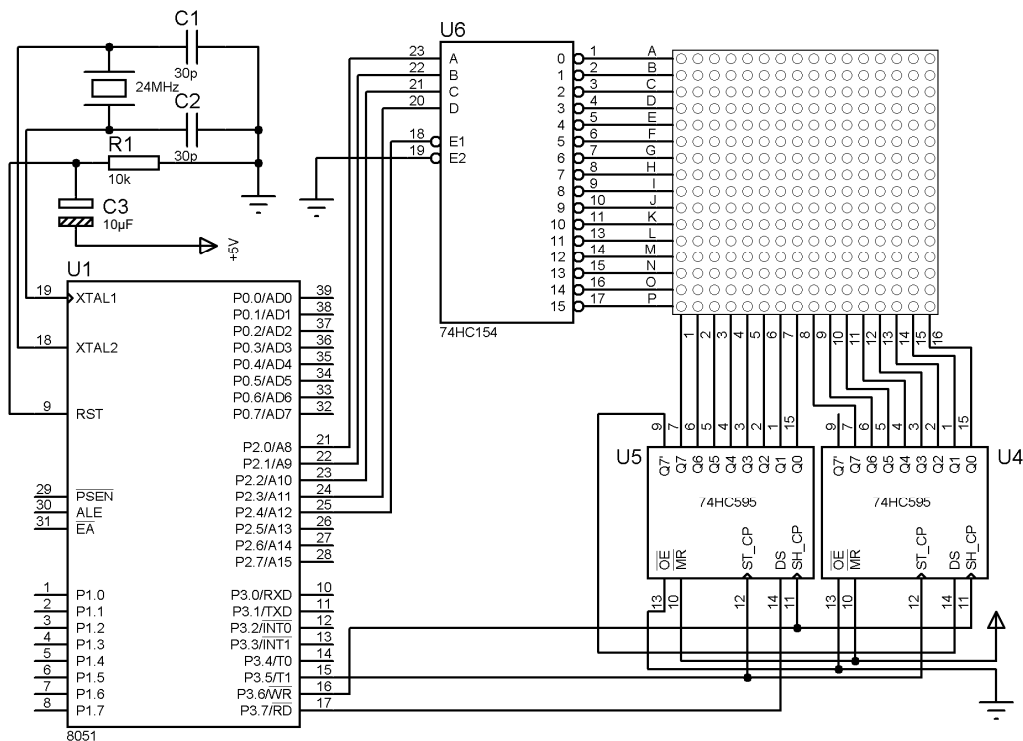


图9.5 16×16点阵LED显示屏的硬件电路图

屏的列驱动。74HC595具有1个8位串入并出的移位寄存器和1个8位输出锁存器，并且移位寄存器和输出锁存器的控制信号各自独立，可以实现在显示本行各列数据的同时，传送下一行的列数据，达到重叠处理的目的。单片机采用24MHz的振荡器以便获得较高的刷新频率，使显示更为稳定。

9.2.3 软件程序设计

显示屏软件的主要功能是向屏体提供显示数据，并产生各种控制信号，使屏幕按设计要求进行显示。利用T0定时中断进行显示刷新，1/16扫描显示屏刷新率（帧频）的计算公式为

$$\text{刷新率（帧频）} = f_{\text{osc}} / 16 \times 12 \quad (65536 - T_0 \text{初值})$$

式中， f_{osc} 为系统时钟频率，即晶振频率。

显示驱动程序从显示缓冲区读取各行显示数据，发送给74HC595的移位寄存器，为了消除在切换行显示数据时产生拖尾现象，先要关闭显示屏，等数据打入输出锁存器之后，再输出新的一行并打开显示屏。

例9-2 点阵LED显示屏驱动程序源文件清单。

```
#include "reg51.h"
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int
#define BLKN 2 //列锁存器数

sbit DS = P3^7; //串行数据输入
sbit SH_CP = P3^6; //移位时钟脉冲
sbit ST_CP = P3^5; //输出锁存器控制脉冲
sbit G_74154 = P2^4; //显示允许控制信号端口

uchar data dispram[32]; //显示缓冲区
uchar temp;
uchar code Bmp[][32]={ //显示数据
    0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
    0xff,0xff,0xff, //黑
    0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
    0xff,0xff,0xff, //屏
    0xf7,0xff,0xf7,0xef,0xf7,0xcf,0xf7,0xbf,0xf7,0x7f,0xf6,0xff,0xf7,
    0xfb,0x0,0x1, //长
    0xf6,0xff,0xf6,0xff,0xf7,0x7f,0xf7,0xbf,0xf7,0xdf,0xf6,0xe3,0xf1,
    0xf7,0xf7,0xff,0xbf,0xff,0xcf,0xf7,0xe8,0x3,0xff,0xbf,0x7f,0xbf,0x9f,
    0xbf,0xdf,0xbf,0xf7,0xbf, //江
    0xef,0xbf,0xdf,0xbf,0x1f,0xbf,0xdf,0xbf,0xdf,0xbb,0xd0,0x1,0xdf,0xff,
    0xdf,0xff,0xfe,0xff,0xfe,0xff,0xfe,0xff,0xfe,0xff,0xfe,0xfb,0x0,0x1,
    0xfe,0xff,0xfd,0x7f, //大
    0xfd,0x7f,0xfd,0xbf,0xfb,0xbf,0xfb,0xdf,0xf7,0xef,0xef,0xf1,0x9f,
    0xfb,0xff,0xff,0xdd,0xf7,0xee,0xf7,0xee,0xef,0xff,0xdf,0x80,0x1,0xbf,
    0xfd,0x7f,0xfb,0xe0,0x1f, //学
```

```

    0xff,0xbf,0xfe,0x7b,0x0,0x1,0xfe,0xff,0xfe,0xff,0xfe,0xff,0xfa,0xff,
    0xfd,0xff
};

/***** 延时函数 *****/
void delay(uint dt){
    uchar bt;
    for(;dt;dt--)
        for(bt=0;bt<255;bt++);
}

/***** 写入74HC595函数 *****/
void WR_595(void){
    uchar x;
    for (x=0;x<8;x++){
        temp=temp<<1;
        DS=CX;
        SH_CP=1;          //上升沿移位
        _nop_();
        _nop_();
        SH_CP=0;
    }
}

/***** T0中断服务函数 *****/
void leddisplay(void) interrupt 1 using 1{
    register unsigned char i,j=BLKN;
    TH0 = 0xF8;           //每秒62.5帧@24MHz
    TL0 = 0x30;
    i=P2;                 //读取当前显示的行号
    i++;                  //行号加1，屏蔽高4位
    do{
        j--;
        temp = dispram[i*BLKN+j];
        WR_595();
    }while(j);
    G_74154=1;           //关闭显示
    P2 &= 0xf0;          //行号端口清零
    ST_CP = 1;           //上升沿将数据送到输出锁存器
    P2 |= i;             //写入行号
    ST_CP = 0;           //锁存显示数据
    G_74154=0;          //打开显示
}

/***** 主函数 *****/
void main(void){
    uchar i,j,k;

```



```

TMOD = 0x01;          //定时器T0工作方式1
TH0 = 0xF8;           //每秒62.5帧@24MHz
TL0 = 0x30;
G_74154 = 1;          //关闭显示
ST_CP=0;
P2 =0xF0;
IE = 0x82;            //允许定时器T0中断
TR0 = 1;              //启动定时器T0
while(1) {
    for(i=0;i<32;i++){          //黑屏
        dispram[i]= ~Bmp[0][i];
    }
    for(i=1;i<5;i++){          //上滚屏显示
        for(j=0;j<16;j++){
            for(k=0;k<15;k++){
                dispram[k*BLKN]=dispram[(k+1)*BLKN];
                dispram[k*BLKN+1]=dispram[(k+1)*BLKN+1];
            }
            dispram[30]=~Bmp[i][j*BLKN];
            dispram[31]=~Bmp[i][j*BLKN+1];
            delay(100);
        }
        delay(1000);
    }
    delay(1000);
}
}

```

9.3 电子密码锁

9.3.1 功能要求

采用8051单片机设计一个电子密码锁。其密码为6位十进制码。由0~9十个按键输入密码，Enter键确认。当输入密码与预设密码一致时，锁被打开，锁开信号灯点亮；当密码不一致时，要求重新输入，如果3次输入密码不一致，则发出声、光报警。具有密码重置功能，重置密码存入串行EEPROM芯片24C01，掉电后，密码不会丢失。

9.3.2 硬件电路设计

图9.6为电子密码锁的硬件电路图。其核心为8051单片机，控制整个密码锁的全部功能。采用3×4矩阵键盘，用于密码输入、重置和修改，另外还设置了一个初始密码单独按键，系统启动时按下该键，将初始密码设置为012345。显示器采用12864图型液晶模块，该液晶模块显示信息丰富，可为密码锁提供良好的人机交互性。密码输入时不显示密码数字，

而是以“*”代替,提高密码锁的可靠性。若发生3次密码输入错误,则通过蜂鸣器和发光二极管报警。由于要求掉电后重置密码不会丢失,所以重置密码不储存在单片机片内RAM里,而是储存在外面扩展的串行EEPROM芯片24C01中。

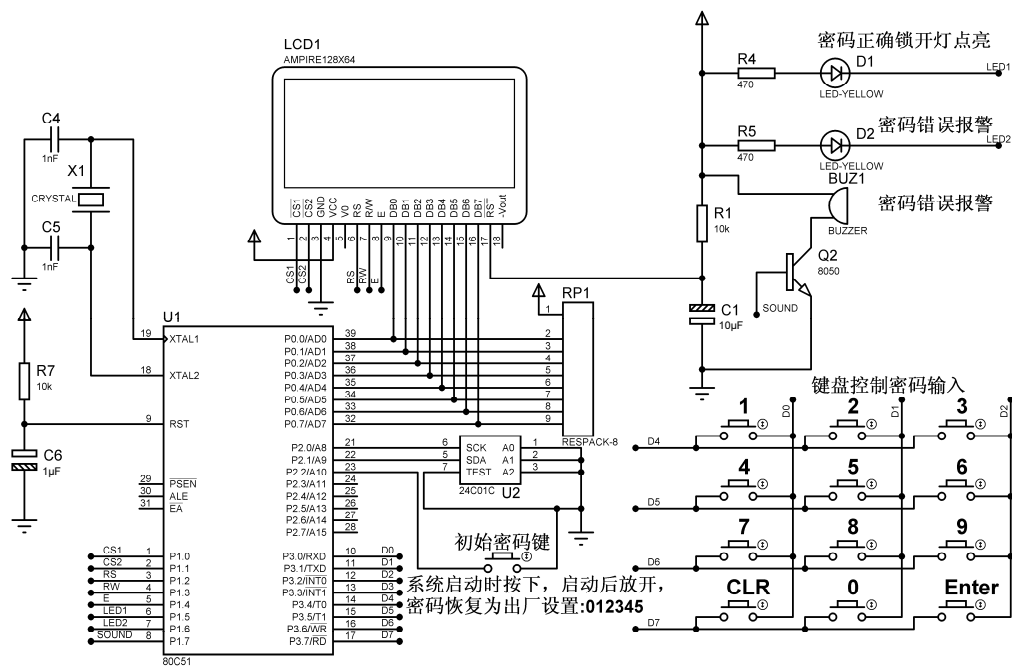


图9.6 电子密码锁的硬件电路图

9.3.3 软件程序设计

例9-3 电子密码锁软件程序。电子密码整个软件程序分模块编写,包括主程序模块main.c、键盘处理程序模块keyinput.h、液晶显示程序模块12864.h和24C01读写程序模块24C01.h等。

(1) 主程序模块main.c文件清单。

```
#include<reg51.h>
#include<keyinput.h>
#include<12864.h>
#include<24C01.h>
#define uchar unsigned char
#define uint unsigned int

sbit LED1=P1^5;
sbit LED2=P1^6;
sbit INIT=P2^2;
sbit SOUND=P1^7;

uchar idata key[6]={0,0,0,0,0,0};
uchar idata iic[6]={0,1,2,3,4,5};
```

```

/***** 密码校验函数 *****/
void press(uchar *s) {
    uchar dat;
    P3=0xf0; //第一位密码
    while(P3==0xf0);
    dat=key_scan();
    if((dat!=0x0a)&&(dat!=0x0b)) {
        *s=dat;
        Left();
        star_12864(star,0x05,16);
    }
    s++;
    P3=0xf0; //第二位密码
    while(P3==0xf0);
    dat=key_scan();
    if((dat!=0x0a)&&(dat!=0x0b)) {
        *s=dat;
        Left();
        star_12864(star,0x05,24);
    }
    s++;
    P3=0xf0; //第三位密码
    while(P3==0xf0);
    dat=key_scan();
    if((dat!=0x0a)&&(dat!=0x0b)) {
        *s=dat;
        Left();
        star_12864(star,0x05,32);
    }
    s++;
    P3=0xf0; //第四位密码
    while(P3==0xf0);
    dat=key_scan();
    if((dat!=0x0a)&&(dat!=0x0b)) {
        *s=dat;
        Left();
        star_12864(star,0x05,40);
    }
    s++;
    P3=0xf0; //第五位密码
    while(P3==0xf0);
    dat=key_scan();
    if((dat!=0x0a)&&(dat!=0x0b)) {
        *s=dat;
        Left();
        star_12864(star,0x05,48);
    }
}

```

```

s++;
P3=0xf0; //第六位密码
while(P3==0xf0);
dat=key_scan();
if((dat!=0x0a)&&(dat!=0x0b)) {
    *s=dat;
    Left();
    star_12864(star,0x05,56);
}
do{P3=0xf0; //键入Enter键继续执行下面语句,否则等待
    while(P3==0xf0);
    dat=key_scan();
}while(dat!=0x0b);
}

/***** 延时10ms函数 *****/
void Delay10ms(void) {
    uint i,j,k;
    for(i=5;i>0;i--)
        for(j=4;j>0;j--)
            for(k=248;k>0;k--);
}

/***** 主函数 *****/
void main() {
    uchar dat;
    uchar i=0,j=0,k;
    uchar x;
    LED1=1; LED2=1; SOUND=0; INIT=1;
    if(INIT==0) { //密码初始化
        x=SendB(iic,0x50,6); //先从I2C器件中读出密码以供下面输入比较
        Delay10ms();
    }
    x=ReadB(iic,0x50,6);
    Init_12864();
    for(i=0;i<50;i++){Delay10ms();}
    do { //若密码不正确,循环执行do{}while()
        LED1=1;
        System(); //显示: 请输入密码
        press(key);
        if((key[0]==iic[0])&&(key[1]==iic[1])&&(key[2]==iic[2])
            &&(key[3]==iic[3])&&(key[4]==iic[4])&&(key[5]==iic[5]))
            //密码比较,若密码正确则进入系统,若密码不正确则显示密码错误,重新输入密码
        {
            true();
            do {
                P3=0xf0; //键入1或2继续执行下面语句,否则等待

```

```

        while(P3==0xf0);
        dat=key_scan();
    }while(dat!=0x01&&dat!=0x02);
    if(dat==1) {                //开锁
        LED1=0; j=0;
        unlock();
        for(i=0;i<100;i++){Delay10ms();}
        continue;
    }
    if(dat==2) {                //修改密码
        do{
            j=0;
            System();
            press(key);
            again();
            press(iic);
            if((key[0]==iic[0])&&
                (key[1]==iic[1])&&(key[2]==iic[2])&&
                (key[3]==iic[3])&&(key[4]==iic[4])&&
                (key[5]==iic[5])){
                succeed();                //修改密码成功
                for(i=0;i<100;i++){Delay10ms();}
                Delay10ms();
                x=SendB(iic,0x50,6);
                Delay10ms();
                x=ReadB(iic,0x50,6);break;
            }
            else {                //修改密码不成功,重新修改
                repeat();
                for(i=0;i<100;i++){Delay10ms();}
            }
        }while(1);
    }
}
else {                //密码不正确,重新输入密码
    j++;
    error();
    if(j==3) {
        for(i=0;i<8;i++) {                //三次密码不正确,报警
            LED2=0; SOUND=1;
            for(k=0;k<5;k++){Delay10ms();}
            LED2=1;
            for(k=0;k<5;k++){Delay10ms();}
        }
        j=0; SOUND=0;
    }
    for(i=0;i<50;i++){Delay10ms();}
}

```

```

    }
    }while(1);
}

```

(2) 键盘处理程序模块keyinput.h文件清单。

```

#include<reg51.h>
#include<absacc.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int

uchar idata com1,com2;

/***** 键盘扫描函数 *****/
uchar key_scan() {
    uchar temp;
    uchar com;
    P3=0xf0;
    if(P3!=0xf0) {
        com1=P3;
        P3=0x0f;
        com2=P3;
    }
    P3=0xf0;
    while(P3!=0xf0);
    temp=com1|com2;
    if(temp==0xee) com=0x01; //密码数字键
    if(temp==0xed) com=0x02;
    if(temp==0xeb) com=0x03;
    if(temp==0xde) com=0x04;
    if(temp==0xdd) com=0x05;
    if(temp==0xdb) com=0x06;
    if(temp==0xbe) com=0x07;
    if(temp==0xbd) com=0x08;
    if(temp==0xbb) com=0x09;
    if(temp==0x7e) com=0x0a;
    if(temp==0x7d) com=0x00;
    if(temp==0x7b) com=0x0b; //Enter键，输入密码结束并确认
    return(com);
}

```

(3) 液晶显示程序模块12864.h文件清单。

```

#include<reg51.h>
#include<absacc.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int

```

```
#define PORT P0
```

```
uchar code Num[]={ //32×32字节的汉字取模，一个汉字72字节
```

```
    0x00,0x00,0x00,0x00,0x10,0x00,0x00,0x08,
    0x00,0x00,0x46,0x00,0x00,0x47,0x00,0xC0,
    0x45,0x00,0xF0,0x64,0x1E,0x7E,0xFE,0x1F,
    0x4E,0x26,0x0C,0x60,0x32,0x06,0x60,0x32,
    0x42,0x00,0x00,0x40,0x30,0x86,0x21,0x70,
    0xFF,0x33,0x20,0x03,0x18,0x03,0xD9,0x0F,
    0xFF,0xF9,0x03,0x06,0x09,0x04,0x20,0x01,
    0x0C,0xB0,0xFF,0x1B,0x1C,0xFF,0x39,0x0C,
    0x00,0x70,0x08,0x00,0x00,0x00,0x00,0x00, //锁
    0x00,0x00,0x00,0x00,0x08,0x00,0x00,0x08,
    0x00,0x00,0x08,0x10,0x00,0x08,0x10,0x10,
    0x0C,0x08,0x10,0x0C,0x0E,0x10,0x84,0x03,
    0xF8,0xFF,0x01,0xF8,0x3F,0x00,0x18,0x06,
    0x00,0x18,0x06,0x00,0x1C,0x06,0x00,0xFC,
    0xFF,0x07,0xFC,0xFF,0xFF,0x0C,0x02,0x00,
    0x0C,0x03,0x00,0x0C,0x03,0x00,0x00,0x03,
    0x00,0x00,0x03,0x00,0x00,0x03,0x00,0x00,
    0x03,0x00,0x00,0x02,0x00,0x00,0x00,0x00, //开
```

```
};
```

```
uchar code Tab[]={ //16×16字节的汉字取模，一个汉字32个字节
```

```
    0x00,0xF8,0x48,0x48,0x48,0x48,0xFF,0x48,
    0x48,0x48,0x48,0xFC,0x08,0x00,0x00,0x00,
    0x00,0x07,0x02,0x02,0x02,0x02,0x3F,0x42,
    0x42,0x42,0x42,0x47,0x40,0x70,0x00,0x00, //电
    0x80,0x80,0x82,0x82,0x82,0x82,0x82,0xE2,
    0xA2,0x92,0x8A,0x86,0x80,0xC0,0x80,0x00,
    0x00,0x00,0x00,0x00,0x00,0x40,0x80,0x7F,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, //子
    0x10,0x4C,0x24,0x04,0xF4,0x84,0x4D,0x56,
    0x24,0x24,0x14,0x84,0x24,0x54,0x0C,0x00,
    0x00,0x01,0xFD,0x41,0x40,0x41,0x41,0x7F,
    0x41,0x41,0x41,0x41,0xFC,0x00,0x00,0x00, //密
    0x02,0x82,0xF2,0x4E,0x43,0xE2,0x42,0xFA,
    0x02,0x02,0x02,0xFF,0x02,0x80,0x00,0x00,
    0x01,0x00,0x7F,0x20,0x20,0x7F,0x08,0x09,
    0x09,0x09,0x0D,0x49,0x81,0x7F,0x01,0x00, //码
    0x80,0x40,0x70,0xCF,0x48,0x48,0x00,0xE2,
    0x2C,0x20,0xBF,0x20,0x28,0xF6,0x20,0x00,
    0x00,0x02,0x02,0x7F,0x22,0x92,0x80,0x4F,
    0x40,0x20,0x1F,0x20,0x20,0x4F,0x80,0x00, //锁
    0x20,0x22,0xEC,0x00,0x20,0x22,0xAA,0xAA,
    0xAA,0xBF,0xAA,0xAA,0xEB,0xA2,0x20,0x00,
    0x00,0x00,0x7F,0x20,0x10,0x00,0xFF,0x0A,
    0x0A,0x0A,0x4A,0x8A,0x7F,0x00,0x00,0x00, //请
```

```

0x88,0x68,0x1F,0xC8,0x0C,0x28,0x90,0xA8,
0xA6,0xA1,0x26,0x28,0x10,0xB0,0x10,0x00,
0x09,0x09,0x05,0xFF,0x05,0x00,0xFF,0x0A,
0x8A,0xFF,0x00,0x1F,0x80,0xFF,0x00,0x00, //输
0x00,0x00,0x00,0x00,0x00,0x01,0xE2,0x1C,
0xE0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x80,0x40,0x20,0x10,0x0C,0x03,0x00,0x00,
0x00,0x03,0x0C,0x30,0x40,0xC0,0x40,0x00, //入
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x33,0x33,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00, //:
0x80,0x40,0x70,0xCF,0x48,0x48,0x48,0x48,
0x7F,0x48,0x48,0x7F,0xC8,0x68,0x40,0x00,
0x00,0x02,0x02,0x7F,0x22,0x12,0x00,0xFF,
0x49,0x49,0x49,0x49,0xFF,0x01,0x00,0x00, //错
0x40,0x42,0xC4,0x0C,0x00,0x40,0x5E,0x52,
0x52,0xD2,0x52,0x52,0x5F,0x42,0x00,0x00,
0x00,0x00,0x7F,0x20,0x12,0x82,0x42,0x22,
0x1A,0x07,0x1A,0x22,0x42,0xC3,0x42,0x00, //误
0x08,0x08,0x0A,0xEA,0xAA,0xAA,0xAA,0xFE,
0xAA,0xAA,0xA9,0xF9,0x29,0x0C,0x08,0x00,
0x40,0x40,0x48,0x4B,0x4A,0x4A,0x4A,0x7F,
0x4A,0x4A,0x4A,0x4B,0x48,0x60,0x40,0x00, //重
0x40,0x44,0x54,0x65,0xC6,0x64,0xD6,0x44,
0x40,0xFC,0x44,0x42,0xC3,0x62,0x40,0x00,
0x20,0x11,0x49,0x81,0x7F,0x01,0x05,0x29,
0x18,0x07,0x00,0x00,0xFF,0x00,0x00,0x00, //新
0x40,0x42,0x44,0xCC,0x00,0x60,0x5E,0x48,
0xC8,0x7F,0xC8,0x48,0x4C,0x68,0x40,0x00,
0x00,0x40,0x20,0x1F,0x20,0x60,0x90,0x8C,
0x83,0x80,0x8F,0x90,0x90,0xD0,0x5C,0x00, //选
0x10,0x10,0x10,0xFF,0x90,0x50,0x82,0x46,
0x2A,0x92,0x2A,0x46,0x82,0x80,0x80,0x00,
0x02,0x42,0x81,0x7F,0x00,0x09,0x08,0x09,
0x09,0xFF,0x09,0x09,0x0C,0x09,0x00,0x00, //择
0x80,0x82,0x82,0x82,0xFE,0x82,0x82,0x82,
0x82,0x82,0xFE,0x82,0x83,0xC2,0x80,0x00,
0x00,0x80,0x40,0x30,0x0F,0x00,0x00,0x00,
0x00,0x00,0xFF,0x00,0x00,0x00,0x00,0x00, //开
0x40,0x20,0xF8,0x07,0xF0,0xA0,0x90,0x4F,
0x54,0x24,0xD4,0x4C,0x84,0x80,0x80,0x00,
0x00,0x00,0xFF,0x00,0x0F,0x80,0x92,0x52,
0x49,0x25,0x24,0x12,0x08,0x00,0x00,0x00, //修
0x04,0xC4,0x44,0x44,0x44,0xFE,0x44,0x20,
0xDF,0x10,0x10,0x10,0xF0,0x18,0x10,0x00,
0x00,0x7F,0x20,0x20,0x10,0x90,0x80,0x40,

```



```

0x21,0x16,0x08,0x16,0x61,0xC0,0x40,0x00, //改
0x00,0x02,0x02,0xF2,0x92,0x92,0x92,0xFE,
0x92,0x92,0x92,0xFA,0x13,0x02,0x00,0x00,
0x04,0x04,0x04,0xFF,0x04,0x04,0x04,0x07,
0x04,0x44,0x84,0x7F,0x04,0x06,0x04,0x00, //再
0x00,0x02,0x04,0x8C,0x40,0x00,0x20,0x18,
0x17,0xD0,0x10,0x50,0x38,0x10,0x00,0x00,
0x02,0x02,0xFF,0x00,0x80,0x40,0x20,0x10,
0x0C,0x03,0x0C,0x10,0x60,0xC0,0x40,0x00, //次
0x04,0x84,0xE4,0x9C,0x84,0xC6,0x24,0xF0,
0x28,0x27,0xF4,0x2C,0x24,0xF0,0x20,0x00,
0x01,0x00,0x7F,0x20,0x20,0xBF,0x40,0x3F,
0x09,0x09,0x7F,0x09,0x89,0xFF,0x00,0x00, //确
0x40,0x42,0x44,0xCC,0x00,0x00,0x00,0x00,
0xC0,0x3F,0xC0,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x3F,0x90,0x48,0x30,0x0E,
0x01,0x00,0x01,0x0E,0x30,0xC0,0x40,0x00, //认
0x00,0x00,0xF8,0x88,0x88,0x88,0x88,0x08,
0x7F,0x88,0x0A,0x0C,0x08,0xC8,0x00,0x00,
0x40,0x20,0x1F,0x00,0x08,0x10,0x0F,0x40,
0x20,0x13,0x1C,0x24,0x43,0x80,0xF0,0x00, //成
0x08,0x08,0x08,0xF8,0x0C,0x28,0x20,0x20,
0xFF,0x20,0x20,0x20,0x20,0xF0,0x20,0x00,
0x08,0x18,0x08,0x0F,0x84,0x44,0x20,0x1C,
0x03,0x20,0x40,0x80,0x40,0x3F,0x00,0x00, //功
0x00,0x00,0x00,0x00,0x00,0x00,0x08,0xF8,
0xFC,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x3F,
0x3F,0x20,0x00,0x00,0x00,0x00,0x00,0x00, //1
0x00,0x00,0x00,0x00,0x30,0x38,0x0C,0x04,
0x04,0x0C,0xF8,0xF0,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x20,0x30,0x38,0x2C,
0x26,0x23,0x21,0x38,0x00,0x00,0x00,0x00, //2
};
uchar code star[]={0x00,0x08,0x2A,0x1C,0x1C,0x2A,0x08,0x00,}; // *号
sbit CS1=P1^0;
sbit CS2=P1^1;
sbit RS=P1^2;
sbit RW=P1^3;
sbit E=P1^4;
sbit bflag=P0^7;

/***** 选左半屏函数 *****/
void Left() {
    CS1=0; CS2=1;
}

```

```

/***** 选右半屏函数 *****/
void Right() {
    CS1=1; CS2=0;
}

/***** 判忙函数 *****/
void Busy_12864() {
    do{
        E=0; RS=0; RW=1;
        PORT=0xff;
        E=1; E=0;
    }while(bflag);
}

/***** 命令写入函数 *****/
void Wreg(uchar c) {
    Busy_12864();
    RS=0; RW=0;
    PORT=c;
    E=1; E=0;
}

/***** 数据写入函数 *****/
void Wdata(uchar c) {
    Busy_12864();
    RS=1; RW=0;
    PORT=c;
    E=1; E=0;
}

/***** 首页函数 *****/
void Pagefirst(uchar c) {
    uchar i;
    i=c;
    c=i|0xb8;
    Busy_12864();
    Wreg(c);
}

/***** 首行函数 *****/
void Linefirst(uchar c) {
    uchar i;
    i=c;
    c=i|0x40;
    Busy_12864();
    Wreg(c);
}

```

```

/***** 清屏函数 *****/
void Ready_12864() {
    uint i,j;
    Left();
    Wreg(0x3f);
    Right();
    Wreg(0x3f);
    Left();
    for(i=0;i<8;i++){
        Pagefirst(i);
        Linefirst(0x00);
        for(j=0;j<64;j++){
            Wdata(0x00);
        }
    }
    Right();
    for(i=0;i<8;i++){
        Pagefirst(i);
        Linefirst(0x00);
        for(j=0;j<64;j++){
            Wdata(0x00);
        }
    }
}

/***** 16×16汉字显示函数 *****/
void Display(uchar *s,uchar page,uchar line) {
    uchar i,j;
    Pagefirst(page);
    Linefirst(line);
    for(i=0;i<16;i++){
        Wdata(*s);
        s++;
    }
    Pagefirst(page+1);
    Linefirst(line);
    for(j=0;j<16;j++){
        Wdata(*s);
        s++;
    }
}

/***** 24×24汉字显示函数 *****/
void Display_32(uchar *s,uchar page,uchar line) {
    uchar i,j;
    for(i=0;i<24;i++){

```

```

        for(j=0;j<3;j++){
            Pagefirst(page+j);
            Linefirst(line+i);
            Wdata(*s);
            s++;
        }
    }
}

/***** 星号显示函数 *****/
void star_12864(uchar *s,uchar page,uchar line)    {
    uchar i;
    Pagefirst(page);
    Linefirst(line);
    for(i=0;i<8;i++){
        Wdata(*s);
        s++;
    }
}

/***** 画线函数 *****/
void point_12864(uchar page,uchar line)    {
    uchar i;
    Pagefirst(page);
    Linefirst(line);
    for(i=0;i<56;i++){
        Wdata(0x1e);
    }
}

/***** 初始化函数 *****/
void Init_12864(){
    Ready_12864();
    Left();
    point_12864(0x03,8);
    Display(Tab,0x04,16);
    Display(Tab+32,0x04,32);
    Display(Tab+64,0x04,48);
    Right();
    point_12864(0x03,0);
    Display(Tab+96,0x04,0);
    Display(Tab+128,0x04,16);
}

/***** 显示请输入密码函数 *****/
void System(){
    Ready_12864();

```

```

    Left();
    Display(Tab+160,0x02,16);
    Display(Tab+192,0x02,32);
    Display(Tab+224,0x02,48);
    point_12864(0x04,8);
    Right();
    Display(Tab+64,0x02,0);
    Display(Tab+96,0x02,16);
    Display(Tab+256,0x02,32);
    point_12864(0x04,0);
}

/***** 显示密码错误函数 *****/
void error(){
    Ready_12864();
    Left();
    Display(Tab+64,0x02,32);
    Display(Tab+96,0x02,48);
    Display(Tab+352,0x04,16);
    Display(Tab+384,0x04,32);
    Display(Tab+192,0x04,48);
    Right();
    Display(Tab+288,0x02,0);
    Display(Tab+320,0x02,16);
    Display(Tab+224,0x04,0);
    Display(Tab+64,0x04,16);
    Display(Tab+96,0x04,32);
}

/***** 显示选择1 开锁, 2 修改密码函数 *****/
void true(){
    Ready_12864();
    Left();
    Display(Tab+160,0x00,0);
    Display(Tab+416,0x00,16);
    Display(Tab+448,0x00,32);
    Display(Tab+256,0x00,48);
    Display(Tab+768,0x03,0);
    Display(Tab+480,0x03,16);
    Display(Tab+128,0x03,32);
    Display(Tab+800,0x06,0);
    Display(Tab+512,0x06,16);
    Display(Tab+544,0x06,32);
    Display(Tab+64,0x06,48);
    Right();
    Display(Tab+96,0x06,0);
}

```

```
/****** 显示开锁画面函数 *****/
void unlock() {
    Ready_12864();
    Left();
    Display_32(Num, 0x03, 20);
    point_12864(0x02, 8);
    point_12864(0x06, 8);
    Right();
    Display_32(Num+72, 0x03, 20);
    point_12864(0x02, 0);
    point_12864(0x06, 0);
}

/****** 显示请再次输入密码函数 *****/
void again() {
    Ready_12864();
    Left();
    Display(Tab+160, 0x00, 0);
    Display(Tab+576, 0x00, 16);
    Display(Tab+608, 0x00, 32);
    Display(Tab+192, 0x00, 48);
    Right();
    Display(Tab+224, 0x00, 0);
    Display(Tab+64, 0x00, 16);
    Display(Tab+96, 0x00, 32);
    Display(Tab+256, 0x00, 48);
}

/****** 显示密码确认错误函数 *****/
void repeat() {
    Ready_12864();
    Left();
    Display(Tab+64, 0x02, 16);
    Display(Tab+96, 0x02, 32);
    Display(Tab+640, 0x02, 48);
    Display(Tab+160, 0x04, 16);
    Display(Tab+352, 0x04, 32);
    Display(Tab+384, 0x04, 48);
    Right();
    Display(Tab+672, 0x02, 0);
    Display(Tab+288, 0x02, 16);
    Display(Tab+320, 0x02, 32);
    Display(Tab+512, 0x04, 0);
    Display(Tab+544, 0x04, 16);
    Display(Tab+64, 0x04, 32);
    Display(Tab+96, 0x04, 48);
}
```

```

}

/***** 显示修改密码成功函数 *****/
void succeed() {
    Ready_12864();
    Left();
    Display(Tab+512, 0x02, 16);
    Display(Tab+544, 0x02, 32);
    Display(Tab+64, 0x02, 48);
    Right();
    Display(Tab+96, 0x02, 0);
    Display(Tab+704, 0x02, 16);
    Display(Tab+736, 0x02, 32);
}

```

(4) 24C01读写模块24C01.h文件清单。

```

#include<reg51.h>
#include<absacc.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int
#define AddWr 0xa0
#define AddRd 0xa1
#define _Nop _nop_

bit ack;
sbit SDA=P2^1;
sbit SCL=P2^0;

/***** I2C器件启动函数 *****/
void Start() {
    SDA=1;
    _Nop();
    SCL=1;
    _Nop(); _Nop(); _Nop(); _Nop(); _Nop();
    SDA=0;
    _Nop(); _Nop(); _Nop(); _Nop();
    SCL=0;
    _Nop(); _Nop();
}

/***** I2C器件停止函数 *****/
void Stop() {
    SDA=0;
    _Nop();
    SCL=1;
    _Nop(); _Nop(); _Nop(); _Nop(); _Nop();
}

```

```

        SDA=1;
        _Nop(); _Nop(); _Nop(); _Nop(); _Nop();
    }

    /***** 检查I2C器件的回复函数 *****/
    void Cack(bit a){
        if(a==0) SDA=0;
        else SDA=1;
        _Nop(); _Nop(); _Nop();
        SCL=1;
        _Nop(); _Nop(); _Nop(); _Nop(); _Nop();
        SCL=0;
        _Nop(); _Nop();
    }

    /***** I2C器件的单字节写入函数 *****/
    void Send(uchar c){ //向I2C器件写入一个字节, 若有回复, ack=1
        uchar i;
        for(i=0;i<8;i++){
            if(c&0x80) SDA=1;
            else SDA=0;
            _Nop();
            SCL=1;
            _Nop(); _Nop(); _Nop(); _Nop(); _Nop();
            SCL=0;
            c=c<<1;
        }
        _Nop(); _Nop();
        SDA=1;
        _Nop(); _Nop();
        SCL=1;
        _Nop(); _Nop(); _Nop();
        if(SDA==1) ack=0;
        else ack=1;
        SCL=0;
        _Nop(); _Nop();
    }

    /***** I2C器件的多字节写入函数 *****/
    bit SendB(uchar *s, uchar Address, uchar Number){
        uchar i; //向I2C器件发送多个字节, 成功返回1
        Start();
        Send(AddWr);
        if(ack==0) return(0);
        Send(Address);
        if(ack==0) return(0);
        for(i=0;i<Number;i++){

```



```

        Send(*s);
        if(ack==0) return(0);
        s++;
    }
    Stop();
    return(1);
}

/***** I2C器件的单字节读取函数 *****/
uchar Read() {
    //从I2C器件读一个字节的內容并返回所读的数据
    uchar temp;
    uchar i;
    temp=0;
    SDA=1;
    for(i=0;i<8;i++){
        _Nop();
        SCL=0;
        _Nop(); _Nop(); _Nop(); _Nop(); _Nop();
        SCL=1;
        _Nop(); _Nop();
        temp=temp<<1;
        if(SDA==1) temp++;
        _Nop(); _Nop();
    }
    SCL=0;
    _Nop(); _Nop();
    return(temp);
}

/***** I2C器件的多字节读取函数 *****/
bit ReadB(uchar *s,uchar Address,uchar Number){
    uchar i;
    //从I2C器件读出多个字节,并将所读的数据存入数组
    Start();
    Send(AddWr);
    if(ack==0) return(0);
    Send(Address);
    if(ack==0) return(0);
    Start();
    Send(AddRd);
    if(ack==0) return(0);
    for(i=0;i<Number;i++){
        *s=Read();
        Cack(0);
        s++;
    }
    *s=Read();
    Cack(1);
    Stop();
}

```

```
return(1);
```

```
}
```

9.4 DS18B20多点温度监测系统

9.4.1 功能要求

采用8051单片机和数字温度传感器DS18B20设计一个多点温度监测系统，测温范围为 $-55\sim 128^{\circ}\text{C}$ ，测量精度为 0.1°C ，采用8位7段LED数码管作为显示器，分时显示当前各点温度监测值和每个DS18B20的ROM序列号，配置两个按键，由按键设定显示内容。系统启动后，默认状态为循环显示各监测点的当前温度值，按一次K2键切换到循环显示各个DS18B20的ROM序列号，再按一次K2键恢复到默认状态。在默认状态下按一次K1键，切换到由K2键选择显示各监测点当前温度值，再按一次K1键恢复到默认状态。

9.4.2 硬件电路设计

多点温度监测系统硬件电路如图9.7所示，主要包括8051单片机、4个DS18B20温度传感器、LED数码管显示器等。

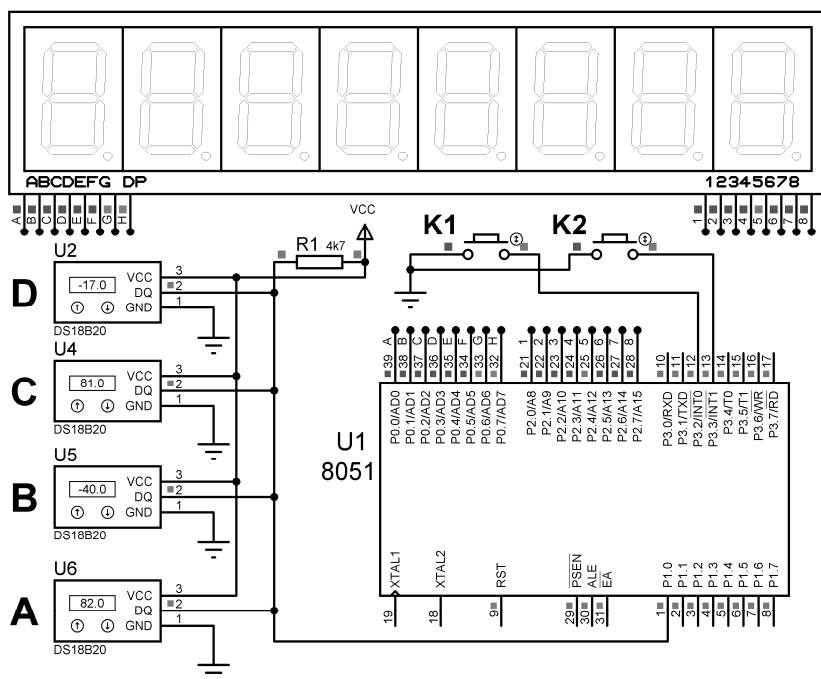


图9.7 DS18B20多点温度监测系统

DS18B20是一种新型数字温度传感器，采用独特的单线接口方式，仅需一个端口引脚来发送和接收信息，在单片机和DS18B20之间仅需一条数据线和一条地线进行接口。DS18B20采用TO—92或8脚SOIC封装，引脚排列如图9.8所示，各引脚功能见表9.1。

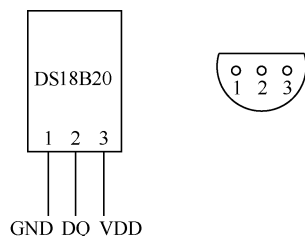


图9.8 DS18B20引脚排列

表9.1 DS18B20各引脚功能

引 脚	功 能
GND	地
DQ	数据输入/输出引脚
VDD	外部供电电源引脚

DS18B20内部有三个主要数字部件：64位激光ROM、温度传感器、非易失性温度报警触发器TH和TL。DS18B20可以采用寄生电源方式工作，从单总线上汲取能量，在信号线处于高电平期间把能量储存在内部电容里，在信号线处于低电平期间利用电容上的电能工作，直到高电平到来再给寄生电源（电容）充电。DS18B20也可用外部3~5.5V电源供电。这两种供电方式的电路如图9.9所示。

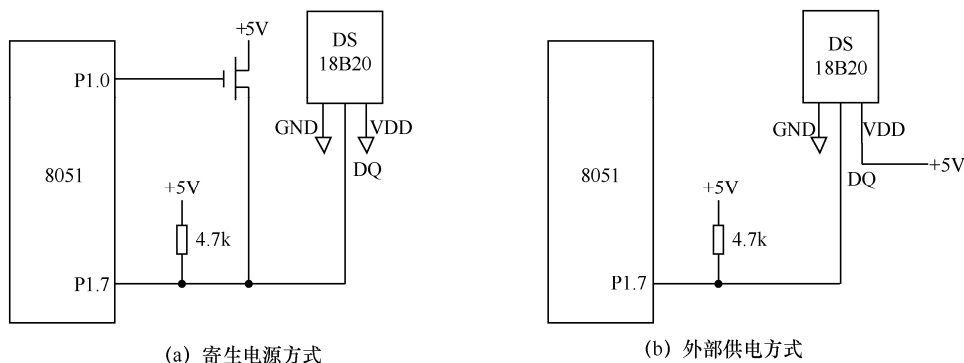


图9.9 DS18B20的供电方式

采用寄生电源方式时，VDD引脚必须接地。另外，为了得到足够的工作电流，应给单片机的I/O口线提供一个强上拉，一般可以使用一个场效应管将I/O口线直接拉到电源上。采用外部供电方式时可以不使用强上拉，但外部电源要处于工作状态，GND引脚不得悬空。温度高于100℃时，不推荐使用寄生电源，应采用外部电源供电。

DS18B20依靠单线方式进行通信，必须先建立ROM操作协议，才能进行存储器和控制操作。单片机必须先提供下面5个ROM操作命令之一：

- 读出ROM，代码为33H，用于读出DS18B20的序列号，即64位激光ROM代码；
- 匹配ROM，代码为55H，用于辨识（或选中）某一特定的DS18B20进行操作；
- 搜索ROM，代码为F0H，用于确定总线上的节点数及所有节点的序列号；
- 跳过ROM，代码为CCH，命令发出后，系统将对所有的DS18B20进行操作，通常用于启动所有的DS18B20进行转换，或系统中仅有一个DS18B20时。

- 报警搜索，代码为ECH，用于鉴别和定位系统中超出程序设定的报警温度界限的节点。

这些命令对每个器件的激光ROM部分进行操作，在单总线上挂有多个器件时，可以区分出各个器件。单片机在发出ROM操作命令之后，紧接着发出存储器操作命令，即可启动温度测量。DS18B20内部存储器映像如图9.10所示。

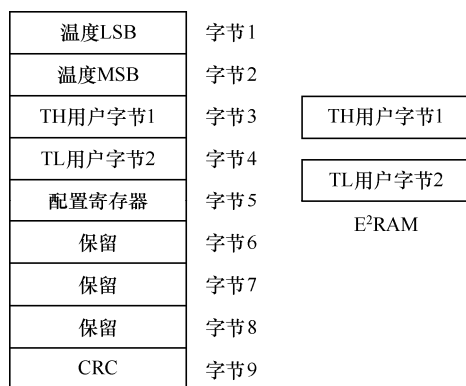


图9.10 DS18B20的存储器映像

存储器由一个高速暂存器和一个存储高低温报警触发值TH和TL的非易失性电可擦除E²RAM组成。在单总线上通信时，暂存器帮助确保数据的完整性。数据先被写入暂存器，并可被读回。数据经过校验后，用一个复制暂存器命令把数据传到非易失性电可擦除E²RAM中。这一过程可确保更改存储器时数据的完整性。

高速暂存器的头两个字节为实测温度值，低字节在前，高字节在后；第3和第4字节是用户设定温度报警值TH和TL的拷贝，是易失的，每次上电时被刷新；第5个字节为配置寄存器，其内容用于确定温度值的数字转换分辨率。DS18B20工作时按此寄存器中的分辨率将温度转换为相应精度的数值。

配置寄存器各位的分布为

D7	D6	D5	D4	D3	D2	D1	D0
TM	R1	R0	1	1	1	1	1

其中，TM为测试模式位，用于设定DS18B20为工作模式还是为测试模式，出厂时，TM位被设置为0，用户一般不要改动；R1和R0用于设定温度转换的精度分辨率，见表9.2；其余低5位全为1。DS18B20温度转换时间较长，而且设定的分辨率越高，所需转换时间越长，在实际应用中要根据具体情况权衡考虑。

表9.2 DS18B20的分辨率设定

R1	R0	分辨率/位	温度最大转换时间（毫秒）
0	0	9	93.75
0	1	10	187.5
1	0	11	375
1	1	12	750

高速暂存器的第6、7、8字节保留未用，读出值为全1。第9字节为前面8个字节的CRC校验码，用于保证数据通信的正确性。

DS18B20提供了如下存储器操作命令：

① 温度转换，代码为44H，用于启动DS18B20进行温度测量。温度转换命令被执行后，DS18B20保持等待状态，如果主机在这条命令之后跟着发出读时间隙，而DS18B20又忙于进行温度转换的话，DS18B20将在总线上输出“0”。若温度转换完成，则输出“1”。如果使用寄生电源，则主机必须在发出这条命令后立即启动强上拉，并保持750ms，在这段时间内，单总线上不允许进行任何其他操作。

② 读暂存器，代码为BEH，用于读取暂存器中的内容。从字节0开始最多可以读取9个字节，如果不想读完所有字节，则主机可以在任何时间发出复位命令来中止读取。

③ 写暂存器，代码为4EH，用于将数据写入到DS18B20暂存器的地址2和地址3（TH和TL字节），可以在任何时刻发出复位命令来中止写入。

④ 复制暂存器，代码为48H，用于将暂存器的内容复制到DS18B20的非易失性E²RAM中，即把温度报警值存入非易失性存储器里。如果主机在这条命令之后跟着发出读时间隙，而DS18B20又正在忙于把暂存器的内容复制到E²RAM存储器，DS18B20就会输出一个“0”。如果复制结束，DS18B20则输出“1”。如果使用寄生电源，则主机必须在这条命令发出后立即启动强上拉并最少保持10ms，在这段时间内，单总线上不允许进行任何其他操作。

⑤ 重读E²RAM，代码为B8H，用于将存储在非易失性E²RAM中的内容重新读入到暂存器（温度报警TH和TL字节）中。这种复制操作在DS18B20上电时自动执行，这样器件一上电，暂存器里马上就存在有效的数据了。若在这条命令发出之后发出读时间隙，器件会输出温度转换忙的标志：“0”代表忙，“1”代表完成。

⑥ 读电源，代码为B4H，用于将DS18B20的供电方式信号发送到主机。若在这条命令发出之后发出读时间隙，则DS18B20将返回它的供电模式：“0”代表寄生电源，“1”代表外部电源。

通过单总线端口访问DS18B20的过程如下：

- 初始化；
- ROM操作命令；
- 存储器操作命令；
- 数据处理。

DS18B20需要严格的时序协议以确保数据的完整性。协议包括几种单总线信号：复位脉冲、存在脉冲、写0、写1、读0和读1。所有这些信号，除了存在脉冲外，都是由主机发出的。与DS18B20之间的任何通信都需要以初始化开始，初始化包括一个由主机发出的复位脉冲和一个紧跟其后由从机发出的存在脉冲，存在脉冲通知主机，DS18B20已经在总线上且已准备好进行发送和接收数据。

一条温度转换命令启动DS18B20完成一次温度测量，测量结果以二进制补码形式存放在高速暂存器中，占用暂存器的字节1（LSB）和字节2（MSB）。用一条读暂存器内容的存储器操作命令可以把暂存器中的数据读出。所有数据都以低位（LSB）在前的方式进行读/写。数据格式为

LSB 字节

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
-------	-------	-------	-------	----------	----------	----------	----------

MSB字节

S	S	S	S	S	2^6	2^5	2^4
---	---	---	---	---	-------	-------	-------

当符号位S=0时,表示测得的温度值为正,可以直接对测得的二进制数进行计算并转换为十进制数。当符号位S=1时,表示测得的温度值为负,此时测得的二进制数为补码数,要先变成原码数再进行计算。表9.3为DS18B20温度与测量值对应。

表9.3 DS18B20温度与测量值对应表

温度 (°C)	二进制数表示	十六进制数表示
+125	00000111 11010000	07D0H
+85	00000101 01010000	0550H
+25.065	00000001 10010001	0191H
+10.125	00000000 10100010	00A2H
+0.5	00000000 00001000	0008H
0	00000000 00000000	0000H
-0.5	11111111 11110000	FFF8H
-10.125	11111111 01011110	FF5EH
-25.0625	11111110 01101111	FE6FH
-55	11111100 10010000	FC90H

DS18B20完成温度转换后,就把温度测量值 t 与暂存器中TH、TL字节的内容进行比较,若 $t > TH$ 或 $t < TL$,则将DS18B20内部报警标志位置1,并对主机发出的报警搜索命令做出响应,因此可用多只DS18B20进行多点温度循环监测。

9.4.3 软件程序设计

多点温度监测系统软件采用C51编写,在主函数中,首先进行ROM搜索,未检测DS18B20时,显示出错信息。当检测到单总线上存在DS18B20时,执行搜索算法,将每个DS18B20的ROM序列号保存到相应数组中,同时发出温度转换命令和读温度命令,完成温度测量,并将当前的温度监测值送到LED数码管进行显示。

例9-4 多点温度监测系统源程序文件清单。

```
#include <reg51.h>
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int
#define MAXNUM 4    //宏定义单总线上最大可用DS18B20个数

sbit DS=P1^0;        //用P1.0口作为各DS18B20与单片机的I/O口
sbit Key0=P3^2;      //P3.2用作按键0的输入,采用外部中断方式获取按键信号
sbit Key1=P3^3;      //P3.3用作按键1的输入,采用外部中断方式获取按键信号
union{                //定义共用体temp用于存放从DS18B20读入的数据
```

```

    uchar c[2];          //其中c[0]是低地址,存放读取温度数值的高字节
    uint x;              //根据大端格式求,数据高字节存在低地址中,低字节存在高地址中
}temp;                  //x便刚好是读出的温度数值
uchar idata flag;       //温度的正负号标志,为1表示负,为0表示为正
uint cc,xs;             //变量cc中保存温度值整数部分,xs保存温度值小数部分的第一位
uchar idata disbuffer[6]; //LED显示缓存数组
uchar idata ID[4][8]={0}; //用于记录各DS18B20的ROM序列号
uchar idata RomID_temp[8]; //匹配DS18B20时临时记录要匹配DS18B20的序列号
uchar m=0;              //m是轮换显示温度值和序列号的全局标志变量
uchar num=0;            //num记录当前单总线上DS18B20的个数
uchar z=0;              //z是按键标志位,为1时表明有按键按下
uchar a=0;              //a是按键K1的记录变量
uchar b=0;              //b是按键K2的记录变量

/***** 时函数 *****/
void delay(uint i){      //延时i*9.62μs
    uint j;
    for(j=i;j>0;j--);
}

/***** 延时函数 *****/
void delay_2μs(uchar i){ //延时 2*i+5 μs
    while(--i);
}

/***** DS18B20复位函数 *****/
uchar DS_init(void){
    uchar presence;
    DS=0; delay_2μs(250); //根据DS18B20的复位时序,先把总线拉低555μs
    DS=1; delay_2μs(30);  //再释放总线,65μs后读取DS18B20发出的信号
    presence=DS;          //如果复位成功,则presence的值为0;否则为1
    delay_2μs(250);
    return (presence);    //返回0则初始化成功,否则失败
}

/***** 单总线读1字节函数 *****/
uchar read_byte(void){
    uchar i,j,dats=0;
    for(i=1;i<=8;i++){    //作8个循环,读出的8位组成一个字节
        DS=0; _nop_();    //先将总线拉低1μs,
        DS=1; delay_2μs(2); //再释放总线,产生读起始信号,延迟9μs后读取
                             DS18B20的值
        j=DS; delay_2μs(30); //一位读完后,延迟65μs后读下一位
        dats=(j<<7)|(dats>>1); //读出的数据最低位在一个字节的最低位
    }
    return(dats);
}

```

```

/***** 单总线读2位函数 *****/
uchar read_2bit(void) {
    uchar i=0,j=0;
    DS=0;   _nop_();           //先将总线拉低1μs,
    DS=1;   delay_2μs(2);      //再释放总线,产生读起始信号,延迟9μs后读取DS18B20发
                                出的值
    j=DS;   delay_2μs(30);     //一位读完后,延迟65μs后读下一位
    DS=0;   _nop_();
    DS=1;   delay_2μs(2);
    i=DS;   delay_2μs(30);
    i=j*2+i;                    //将读出的两位放到变量i中
    return(i);
}

/***** 单总线写1字节函数 *****/
void write_byte(uchar dat) {
    uchar i;
    for(i=0;i<8;i++){          //作8个循环,写入的8位组成一个字节
        DS=0;                  //先将总线拉低
        DS = dat&0x01;         //向总线上放入要写的值
        delay_2μs(50);         //延迟105μs,以使DS18B20能采样到要写入的值
        DS = 1;                //释放总线,准备写入下一位
        dat>>=1;               //将要写的下一位移到dat的最低位
    }
}

/***** 单总线写1位函数 *****/
void write_bit(bit dat) {
    DS=0;                      //先将总线拉低
    DS=dat;                    //向总线上放入要写的值
    delay_2μs(50);             //延迟105μs,以使DS18B20能采样到要写入的值
    DS = 1;                   //释放总线
}

/***** 显示ROM序列号函数 *****/
void display_ROMID(void) {
    uchar i,p,t,k;
    uint q;
    uchar disbuffer_rom[8];
    uchar codevalue[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xf8,
                        0x80,0x90,0x88,0x83,0xc6,0xa1,0x86,0x8e,0xff,0xbf};
                                //共阳极的字段码
    uchar chocode[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
                                //位选码表

    z=0;                        //z归零,表明没有按键按下
    for(q=0;q<500;q++){        //显示各DS18B20的序号

```



```

        if(z==1) break;           //如果z为1, 有按键按下, 表明要显示不同的内容,
                                   跳出循环
        t=chocode[7];             //取当前的位选码
        P2=t;                     //送出位选码
        t=codevalue[m+10];        //查得显示字符的字段码
        P0=t;                     //送出字段码
        delay(100);
    }
    P2=0x00;                      //关断LED一段时间, 产生闪屏效果
    for(q=0;q<500;q++){
        if(z==1)break;           //如果z为1, 有按键按下, 表明要显示不同的内容,
                                   跳出循环

        delay(100);
    }
    for(k=0;k<8;k=k+4){          //依次显示序列号的低32位或高32位
        if(z==1)break;           //如果z为1, 有按键按下, 表明要显示不同的内容,
                                   跳出循环

        disbuffer_rom[0]=(ID[m][k]&0x0F);
                                   //接下来将序列号放入disbuffer_rom存储
        disbuffer_rom[1]=((ID[m][k]&0xF0)>>4);
        disbuffer_rom[2]=(ID[m][k+1]&0x0F);
        disbuffer_rom[3]=((ID[m][k+1]&0xF0)>>4);
        disbuffer_rom[4]=(ID[m][k+2]&0x0F);
        disbuffer_rom[5]=((ID[m][k+2]&0xF0)>>4);
        disbuffer_rom[6]=(ID[m][k+3]&0x0F);
        disbuffer_rom[7]=((ID[m][k+3]&0xF0)>>4);
        for(q=0;q<250;q++){      //显示序列号的低32位或高32位
            if(z==1)break;        //如果z为1, 有按键按下, 显示不同的
                                   内容, 跳出循环

            for (i=0;i<8;i++){
                if(z==1) break;
                t=chocode[i];      //取当前的位选码
                P2=t;              //送出位选码
                p=disbuffer_rom[i]; //取当前显示的字符
                t=codevalue[p];    //查得显示字符的字段码
                P0=t;              //送出字段码
                delay(40);
            }
        }
        P2=0x00;                  //显示完一轮, 灯灭
        for(q=0;q<500;q++){
            if(z==1)break; //如果z为1, 有按键按下, 显示不同的内容, 跳出循环
            delay(100);
        }
    }
}

```

```

/*****显示出错信息函数 *****/
void display_error(void) {
    uchar i,p,t;
    uint q;
    uchar disbuffer_temp[8]={0,2,8,1,16,0x0f,0x0f,0};
                                                //显示内容为"F 1820"
    uchar codevalue[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xf8,
                        0x80,0x90,0x88,0x83,0xc6,0xa1,0x86,0x8e,0xff,0xbf};
                                                //共阳极的字段码
    uchar chocode[]={0x80,0x40,0x20,0x10,0x08,0x04}; //位选码表
    for(q=0;q<250;q++) {
        for (i=0;i<8;i++) {
            t=chocode[i];           //取当前的位选码
            P2=t;                   //送出位选码
            p=disbuffer_temp[i];    //取当前显示的字符
            t=codevalue[p];         //查得显示字符的字段码
            P0=t;                   //送出字段码
            delay(40);
        }
    }
}

/***** 搜索ROM序列号函数 *****/
uchar search_rom(void) {
    uchar k,l=0,chongtuwei,m,n,a;
    uchar _00web[MAXNUM]={0};
    do {
        DS_init();                //复位所有DS18B20
        write_byte(0xf0);          //单片机发布搜索命令
        for(m=0;m<8;m++) {
            char s=0;              //s用来记录本次循环得到的1个字节(8位) 序列号
            for(n=0;n<8;n++) {
                k=read_2bit();      //读第m*8+n位的原码和反码, 保存在k中
                k=k&0x03;          //屏蔽掉k中其他位的干扰, 为下一步判断作准备
                s>>=1;             //s右移一位, 即把上一次循环得到的位值右移一位,
                                   //执行完一次n为变量的循环, 便得到一个字节的ROM序列号
                if(k==0x01) {       //k为01, 表明读到的数据为0, 即所有器件在这一位都为0
                    write_bit(0);  //对这位记为0
                }
                else if(k==0x02) {  //k为02, 表明读到的数据为1, 即所有器件在这一
                                   位都为1
                    s=s|0x80;       //记录下此位的值, 即s的最高位置1
                    write_bit(1);
                }
                else if(k==0x00) {
                    chongtuwei=m*8+n+1; //记录下这个冲突位发生的位置, 之所以加1
                }
            }
        }
    } while(1);
}

```

```

//是为了让_00web数组中的第一位保持0不变，
//便于判断搜索循环是否结束；
if(chongtuwei>_00web[1]){ //如果冲突位比标志00位的位高，即
//发现了新的冲突位，那么这位写0
    write_bit (0);
    _00web[++1]=chongtuwei; //依次记录位比冲突标志位高的冲突位
}
else if(chongtuwei<_00web[1]){ //如果冲突位比标志00位的位低，
//那么把ID中这位所在的字节右移n位，
//从而得到这位先前已经写过的值，
//如果为0，说明这位先前写的是0，
//那么继续写0，
//如果这位先前写的是1，那么继续写1
    a=(ID[num-1][m]>>n)&0x01;
    s=s|(a<<7); //记录下此位的值
    write_bit(a);
}
else if(chongtuwei==_00web[1]){ //如果冲突位就是标志00位，
//那么s的最高位置1，即这位记为1，
//同时向总线上写1；
//之所以不写0，是因为前面已经写过0，
//再写0，就得不到遍历的效果。

    s=s|0x80;
    write_bit (1);
    l=l-1; //改变标志00位的位置，
//即向前推一个00位，并且是往低位推
}
}
else if(k==0x03) //k为03，表明总线上没有DS18B20，函数结束，同时返回零值
{return(0);}
}
ID[num][m]=s;
}
num++; //DS18B20的个数加1
}while(( _00web[1]!=0) && (num<MAXNUM)); //如果冲突位记录数组已经前推到0值，
//或是DS18B20的数目已经超过最大允许数目，
//就退出循环
return(1); //搜索完毕，返回1值
}

/***** 读取温度函数 *****/
void Read_Temperature_rom(void){
    uchar i;
    DS_init(); //复位DS18B20
    write_byte(0x55); //匹配ROM

```

```

        for(i=0;i<8;i++)           //发出64位ROM编码
            write_byte(RomID_temp[i]);
        write_byte(0x44);           //开始测量温度
        DS_init();                  //复位DS18B20
        write_byte(0x55);           //匹配ROM
        for(i=0;i<8;i++)           //发出64位ROM编码
            write_byte(RomID_temp[i]);
        write_byte(0xBE);           //发读温度命令
        temp.c[1]=read_byte();      //读低字节
        temp.c[0]=read_byte();      //读高字节
    }

    /***** 温度转换函数 *****/
    void Temperature_cov(void){
        if (temp.c[0]>0xf8) { //为负则标志置1, 注意c[0]中放的是读取温度值的高字节
            flag=1;           //还要注意读出的数值一共是12位
            temp.x=~temp.x+1;
        }
        cc=temp.x/16;          //计算温度值的整数部分, 相当于数值乘0.0625再取整数
        xs=temp.x&0x0f;        //取温度值小数部分的第一位
        xs=xs*10;              //这两条语句相当于乘0.625, 得小数位的第一位
        xs=xs/16;              //注意不是乘0.0625
    }

    /***** 定义温度底层显示函数 *****/
    void display(void) {
        uchar codevalue[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xf8,
                               //共阳极LED的字段码
                               0x80,0x90,0x88,0x83,0xc6,0xa1,0x86,0x8e,0xff,0xbf};
        uchar chocode[]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01}; //字段码
        uchar i=0,p,t;
        disbuffer[5]=0x0a+m;      //第6位以字母A,B,C,D显示, 所以加上0x0a
        if (flag==1) disbuffer[4]=0x11; //判断是否显示负号
        else disbuffer[4]=0x10;
        disbuffer[0]=xs;          //放入小数位显示的数值
        disbuffer[1]=(cc%10);      //放入个位要显示的数值
        disbuffer[2]=(cc/10);      //放入十位要显示的数值
        if(disbuffer[2]==0x0a)disbuffer[2]=0x00;
            if(disbuffer[2]==0x0b)disbuffer[2]=0x01;
            if(disbuffer[2]==0x0c)disbuffer[2]=0x02;
        disbuffer[3]=(cc/100);     //放入百位要显示的数值
        for (i=0;i<6;i++){
            t=chocode[i];         //取当前的位选码
            P2=t;                 //送出位选码
            p=disbuffer[i];       //取当前显示的字符
            t=codevalue[p];       //查得显示字符的字段码
            if (i==1) t=t+0x80;   //个位比较特殊, 因为有小数点, 所以要加上0x80
        }
    }

```

```

        P0=t;                //送出字段码
        delay(40);
    }
}

/***** 定义温度上层显示函数 *****/
void diplay_final(void) {
    uint q,r;
    z=0;
    for(q=0;q<8;q++) {        //将DS18B20的序列号放入数组RomID_temp中
        RomID_temp[q]=ID[m][q];
    }
    P2=0x00;
    if(a==0){                 //如果按键K1的标志变量a=0,即进行闪烁显示
        for(q=0;q<3;q++){    //闪烁空隙仍在读取温度
            if(z==1) break;   //如果z为1,有按键按下,表明要显示不同内容,
                               跳出循环

            flag=0;
            Read_Temperature_rom(); //读取双字节温度值
            Temperature_cov();     //温度转换
            for(r=0;r<15;r++)
                delay(1000);
        }
    }
    for(q=0;q<5;q++){        //读取温度并显示
        if(z==1) break;      //如果z为1,有按键按下,表明要显示不同的内容,跳出循环
        flag=0;
        Read_Temperature_rom(); //读取双字节温度
        Temperature_cov();     //温度转换
        for(r=0;r<100;r++)
            display();         //显示温度
    }
}

/***** 外部中断0中断处理函数 *****/
void key_0() interrupt 0 {
    delay(1200);             //延时消抖
    if(Key0==0){             //判断是否K1键按下
        if(a==0) a=1;        //a只可能取0值或1值
        else a=0;
        b=0;                 //同时一旦K1键按下就将b归零
        z=1;                 //将有按键按下标志位z置位
    }
}

/***** 外部中断1中断处理函数 *****/
void key_1() interrupt 2 {

```

```

        delay(1200);          //延时消抖
        if (Key1==0) {        //判断是否K2键按下
            if (a==0) {       //只有a等于0时才让b的值在0或1之间转变
                if (b==0) b=1;
                else b=0;
            }
            z=1;               //将有按键按下标志位z置位
            if (a==1) {       //当a=1时进入固定显示一个DS18B20的温度值的状态
                m++;           //按一次K2键m值加1, 即在不同的DS18B20之间切换显示
                if (m>=num) m=0; //m的值不能超过或等于总线上挂接的DS18B20的数目
            }
        }
    }

    /***** 主函数 *****/
    void main() {
        uchar p;
        delay(30);
        p=search_rom();       //搜索DS18B20, 返回值p为1, 表明总线上存在DS18B20
        if (p==0)              //返回值p为0, 说明总线上没有DS18B20, 显示"F 1820"
            while(1)
                display_error();
        EA=1; EX0=1; IT0=1; EX1=1; IT1=1; //开中断, 允许外部中断0和1, 边沿触发方式
        while(1) {
            if (a==0&&b==0) { //按键K1和K2的状态变量值都为0, 循环显示各点温度值
                diplay_final(); //显示DS18B20的温度值
                if (m<num-1) m++; //为了实现循环显示, 要改变m的值
                else m=0;
                if (z==1) m=0;    //如果有按键按下, 将m清零
            }
            else if (a==0&&b==1) { //按键K1和K2的状态变量值分别为0和1, 循环显示ROM序列号
                display_ROMID(); //显示DS18B20的序列号
                if (m<num-1) m++; //为了实现循环显示, 要改变m的值
                else m=0;
                if (z==1) m=0;    //如果有按键按下, 将m清零
            }
            else diplay_final(); //显示DS18B20的温度值
        }
    }
}

```

9.5 SD卡WAV音频播放器

9.5.1 功能要求

采用8051单片机、SD卡、10位DAC芯片TLC5616及FAT32文件系统, 设计一种能播放

Windows PCM格式WAV音频文件的播放器，音频采样速率为4kHz，分辨率为8位，单声道，音频文件储存在以FAT32格式化的SD卡之内，单片机读取SD卡中的音频信息，经DAC还原成声音信号进行播放，同时通过串行口输出各种提示信息。

9.5.2 硬件电路设计

简单Windows PCM格式的WAV文件是对声音信号按一定频率进行采样所获得的数据，再加上WAV信息头而得到的，如对一路声音信号以4kHz频率采样，采样数值以8位（1字节）方式保存，最终的WAV文件就是单声道4kHz、8位PCM格式的WAV文件。其位速为32b/s。单片机实现音频播放时，只要将WAV文件中的音频数据逐个取出，按4kHz频率送给D/A转换器进行转换，即可还原为音频信号，再通过耳机或喇叭进行声音播放。

图9.11为SD卡音频播放器硬件电路图，可以播放存储在SD卡中的WAV音频文件。

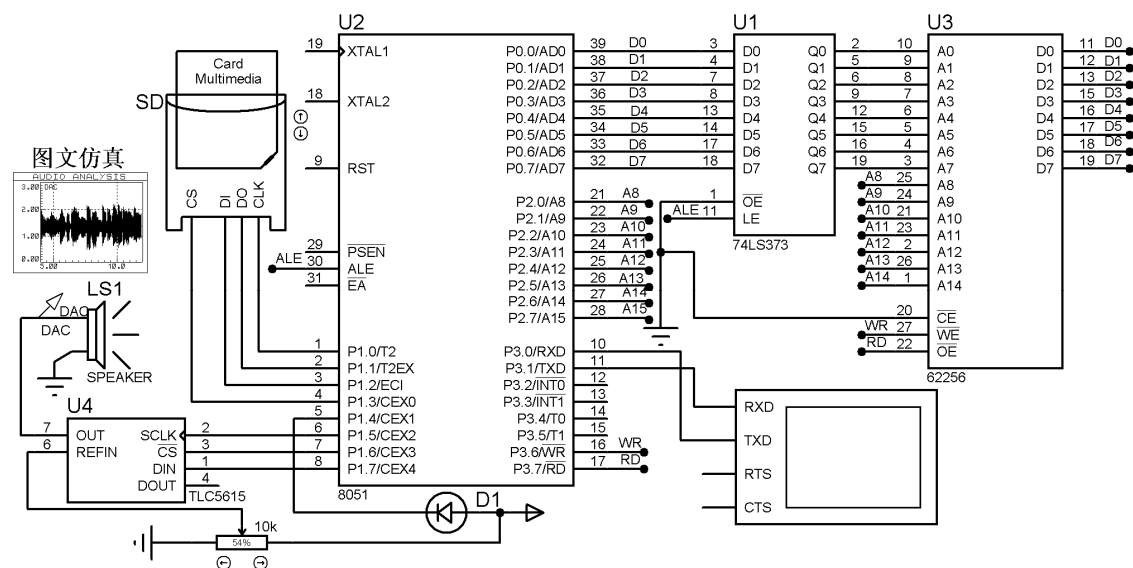


图9.11 SD卡音频播放器硬件电路图

SD卡是一种基于NAND Flash的存储卡。由于它具有安全性高、容量大、体积小、功耗低、非易失性等优点，因此得到广泛的应用。SD卡采用SPI方式与单片机接口，只需要4根信号线：CS（片选）、CLK（时钟）、DI（主机到SD卡的数据信号）和DO（卡到主机的数据信号）。片选信号CS在整个SPI操作过程中必须保持低电平有效；CLK时钟用于同步；DI不但传输数据，还发送命令；DO除了传输数据外，还发送应答信号。将单片机的P1.0、P1.1、P1.2和P1.4分别接到SD卡的CLK、DO、DI和CS端，以软件模拟方式实现SPI工作时序，实现单片机对SD卡的读/写操作。由于对SD卡的一次读/写操作往往需要512个字节，8051单片机片内RAM不够用，因此在外扩展了一片32KB的RAM芯片62256。

D/A转换器采用TLC5615，是一种串行10位DAC芯片，只需要3根串行信号线就可以完成与单片机的接口。单片机的P1.5、P1.6和P1.7分别接到TLC5615芯片的SCLK、 $\overline{\text{CS}}$ 和DIN端。TLC5615采用12位数据序列工作方式，在 $\overline{\text{CS}}$ 为低电平期间，由时钟信号SCLK控制串行

数据DIN向16位移位寄存器依次输入10位有效数据位和低2位填充位（填充位数据任意），高位在前，低位在后，需要12个SCLK时钟完成一次数据传输，数字量与输出模拟电压呈线性关系。

9.5.3 软件程序设计

（1）SD卡底层读、写驱动

SD卡上电后，默认工作在SD模式，切换到SPI模式的方法是，上电后，保持片选信号CS为低电平并延时大于74个CLK时钟周期的时间，然后发送复位命令CMD0，复位成功（当接收到响应信号0x01）后，再连续发送CMD55和ACMD41，直到接收到响应0x00为止，此时SD卡已经进入SPI模式。

SD卡命令由6个字节，共48位组成，格式见表9.4。

表9.4 SD卡命令格式

位标识	47	46	45: 40	39: 8	7: 1	0
宽度	1	1	6	32	7	1
取值	0	1	×	×	×	1
描述	起始位	传输位	命令序号	命令参数	CRC7	停止位

应用时，可以按以下顺序连续发送6个字节来实现上述SD卡命令格式。

字节1: 0 1 x x x x x x（命令号，由指令标志定义，如CMD39为00100111，即16进制0x27，那么完整的CMD39第一字节为01100111，即0x27+0x40）。

字节2-5:命令参数，有些命令没有参数。

字节6:前7位为CRC（循环冗余校验位），最后一位为停止位1。

不同的SD卡，主机根据功能可支持不同的命令集。SD卡命令共分为12类，分别为class0～class11。其中，class1、class3、class9不支持SPI模式。

Class0（卡识别、初始化等基本命令集）：

- CMD0: 复位SD卡；
- CMD1: 读OCR寄存器；
- CMD9: 读CSD寄存器；
- CMD10: 读CID寄存器；
- CMD12: 停止读多块时的数据传输；
- CMD13: 读 Card_Status 寄存器。

Class2（读卡命令集）：

- CMD16: 设置块的长度；
- CMD17: 读单块；
- CMD18: 读多块，直至主机发送CMD12为止。

Class4（写卡命令集）：

- CMD24: 写单块；
- CMD25: 写多块；

CMD27: 写CSD寄存器。

Class5 (擦除卡命令集):

CMD32: 设置擦除块的起始地址;

CMD33: 设置擦除块的终止地址;

CMD38: 擦除所选择的块。

Class6 (写保护命令集):

CMD28: 设置写保护块的地址;

CMD29: 擦除写保护块的地址;

CMD30: 查询SD卡写保护位的状态。

class7 (卡的锁定, 解锁功能命令集)。

class8 (申请特定命令集)。

class10~11 (保留)。

SD卡的所有操作都必须由命令完成, 通过向SD卡发送相关命令并读取相应的响应来实现对SD卡的控制。在对SD卡进行读/写之前, 先要进行初始化操作, 这是确保SD卡能在SPI模式下进行正常数据读/写的前提。SD卡基本读/写操作的命令有: 数据块读命令(CMD17)、多数据块读命令(CMD18)、数据块写命令(CMD24)、多数据块写命令(CMD25)等。需要注意的是, 在发送使SD卡空闲命令(CMD0)之前应至少等待74个CLK时钟周期, 确保SD卡进入SPI模式。初始化完成之后, 如果采用默认的块读/写长度512字节, 则可以直接进行SD卡的读/写; 也可以采用CMD16命令来设置SD卡的块读取长度, 可以是1~512字节之间的任意值。无论读或写操作, 都要求在读/写命令发出后有1个字节的数据起始令牌FEH, 数据传输结束后, 有两个字节的CRC循环冗余校验编码。

例9-5 SD卡接口底层驱动读/写程序清单。

```
#include <reg51.h>
#include <stdio.h>
#define CR 0x0D
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long

sbit SD_CS   = P1^4;
sbit SD_CLK  = P1^0;
sbit SD_DO   = P1^1;
sbit SD_DI   = P1^2;
uchar xdata buf1[512];
uchar xdata buf2[30];

/***** 写一字节到SD卡, 模拟SPI总线方式 *****/
void SdWrite(uchar n){
    uchar i;
    for(i=8;i;i--){
        SD_CLK=0;
        SD_DI=(n&0x80);
```

```

        n<=1;
        SD_CLK=1;
    }
    SD_CLK=0;
}

/***** 从SD卡读一字节，模拟SPI总线方式 *****/
uchar SdRead() {
    uchar n,i;
    for(i=8;i;i--){
        SD_CLK=0;
        SD_CLK=1;
        n<=1;
        if(SD_DO) n|=1;
    }
    return n;
}

/***** 检测SD卡的响应 *****/
uchar SdResponse() {
    uchar i,response;
    while(i<=8){
        response = SdRead();
        if(response==0x00) break;
        if(response==0x01) break;
        i++;
    }
    return response;
}

/***** 发命令到SD卡 *****/
void SdCommand(uchar command, unsigned long argument, uchar CRC){
    SdWrite(command|0x40);
    SdWrite(((uchar *)&argument)[0]);
    SdWrite(((uchar *)&argument)[1]);
    SdWrite(((uchar *)&argument)[2]);
    SdWrite(((uchar *)&argument)[3]);
    SdWrite(CRC);
}

/***** 初始化SD卡 *****/
uchar SdInit(void){
    int delay=0, trials=0;
    uchar i;
    uchar response=0x01;
    SD_CS=1;
    for(i=0;i<=9;i++) SdWrite(0xff);

```

```

        SD_CS=0;
        SdCommand(0x00,0,0x95);
        response=SdResponse();
        if(response!=0x01) return 0;
        while(response==0x01){
            SD_CS=1;
            SdWrite(0xff);
            SD_CS=0;
            SdCommand(0x01,0x00,0xff);
            response=SdResponse();
        }
        SD_CS=1;
        SdWrite(0xff);
        return 1;
    }

    /***** 往SD卡指定地址写扇区，一次最多512字节 *****/
    uchar SD_Write_Sector(uchar *Block, ulong address,int len){
        uint count;
        uchar dataResp;
        printf("SD_Write_block");
        SD_CS=0;
        SdCommand(0x18,address,0xff);
        if(SdResponse()==00){
            SdWrite(0xff);
            SdWrite(0xff);
            SdWrite(0xff);
            SdWrite(0xfe);
            for(count=0;count<len;count++) SdWrite(*Block++);
            for(;count<512;count++) SdWrite(0);
            SdWrite(0xff); //两字节CRC校验，为0xFFFF 表示不考虑CRC
            SdWrite(0xff);
            dataResp=SdRead();
            while(SdRead()==0);
            dataResp=dataResp&0x0f;
            SD_CS=1;
            SdWrite(0xff);
            if(dataResp==0x0b){
                printf("DATA WAS NOT ACCEPTED BY CARD -- CRC ERROR\n");
                return 0;
            }
            if(dataResp==0x05){
                printf("    Write OK\n");
                return 1;
            }
        }
        printf("Invalid data Response token.\n");
        return 0;
    }

```

```

    }
    printf("Command 0x18 (Write) was not received by the SD.\n");
    return 0;
}

/***** 从SD卡指定地址读扇区，一次最多512字节 *****/
uchar SD_Read_Sector(uchar *Block, ulong address, int len){
    uint count;
    printf("SD_read_block");
    SD_CS=0;
    SdCommand(0x11, address, 0xff);
    if(SdResponse()==00) {
        while(SdRead()!=0xfe);
        for(count=0; count<len; count++) *Block++=SdRead();
        for(; count<512; count++) SdRead();
        SdRead();
        SdRead();
        SD_CS=1;
        SdRead();
        printf("    Read Data as Follows\n");
        return 1;
    }
    printf("Command 0x11 (Read) was not received by the SD.\n");
    return 0;
}

/***** 主函数 *****/
void main(void){
    int i, j; uchar SD;
    SCON=0x52; TMOD=0x20;    // 串行口、定时器初始化
    PCON=0x80; TH1=0x0F3;    // fosc=12MHz, 波特率=4800
    TL1=0x0F3; TCON=0x69;
    SD=SdInit();
    if(SD==0){
        printf("SD Card Error\n");
        while(1);
    }
    for(i=0; i<512; i++) {    // 待写入数据装入缓冲区
        buf1[i]=i;
    }
    SD_Write_Sector (buf1, 0x00, 512); // 写地址为0x00的存储块，大小为512字节
    for(i=0; i<512; i++) {
        buf1[i]=0;
    }
    SD_Read_Sector(buf1, 0x00, 512); // 读地址为0x00的存储块，大小为512字节
    for(i=0; i<0x20; i++) {    // 读取的数据通过串行口输出
        for(j=0; j<0x10; j++){

```

```

        printf("%4bX",buf1[(i<<4)+j]);
    }
    printf("%n");
}

SD_Write_Sector("THE SD READ/WRITE TEST OK!",0x000200,26);
                                //写入字符串
SD_Read_Sector(buf2,0x000200,26);    //读出进行检验
for(i=0;i<1;i++) {
    for(j=0; j<0x10; j++){
        printf("%4bX",buf2[(i<<4)+j]);
    }
    printf("%n");
}
printf("  %s",buf2);
while(1);
}

```

(2) FAT32文件系统

完成SD卡的底层读/写驱动之后,还需要文件系统来完成对SD卡上数据的各种操作,如文件系统初始化、打开文件、读取数据、创建文件、获取文件容量。**znFAT**是由中国人自己开发的一种适用于嵌入式系统应用的FAT32文件系统,具有高效、完备、精巧的特点,可移植性强,能够很好地应用在8051单片机系统中。**znFAT**文件系统占用资源很少,可由使用者根据自己的资源情况灵活配置,提供清晰而强大的对函数模块裁剪的功能,极大地减小最终生成的可执行文件长度;提供数据读取的重定向功能,使读到的数据无须缓冲暂存,直接流向应用目的;支持无限深目录和长文件名,长文件名最大长度可自由配置,并可选择是否使用OEM字符集;提供数据写入的实时模式,写入的任何数据,哪怕只有一个字节,也会立即写入到物理存储器中,防止因恶劣工作环境干扰或其他原因引起的不可预见死机或故障,造成数据丢失;支持*与?通配符,支持多设备、多文件操作,可同时挂载多种存储设备,可在多种存储设备间任意切换。

znFAT的移植非常简单,只要有了较为成熟稳定的存储设备扇区读/写等驱动函数,如SD卡的扇区读/写函数SD_Read_Sector()、SD_Write_Sector(),很容易实现**znFAT**文件系统在8051单片机上的移植,具体移植方法请参阅**znFAT**相关书籍。

(3) 音频信号还原

Windows PCM格式的WAV文件中,除了音频信号数据之外,还包括表9.5的WAV文件头信息,占据44个字节。

表9.5 WAV文件头信息

偏移地址	字节数	数据类型	数据内容
00H~03H	4	char	文件标志(RIFF)
04H~07H	4	long	从下个地址开始到文件尾的总字节数
08H~0BH	4	char	WAV文件标志(WAVE)
0CH~0FH	4	char	波形格式标志(fmt),最后一位空格

(续表)

偏移地址	字节数	数据类型	数据内容
10H~13H	4	int	过滤字节(一般为00000010H)
14H~15H	2	int	格式种类(值为1时,表示数据为线性PCM编码)
16H~17H	2	int	通道数,单声道为1,双声道为2
18H~1BH	4	long	采样频率
1CH~1FH	4	long	波形数据传输速率(每秒平均字节数)
20H~21H	2	整数	DATA数据块长度
22H~23H	2	整数	PCM位宽
24H~27H	4	字符	音频数据标志(data),若WAV文件是由某些软件转换而成,通常包含这部分
28H~2BH	4	长整数	音频数据总长度

将单片芯片内定时器T0设置为16位定时工作方式,定时时间为 $250\mu\text{s}$,每隔 $250\mu\text{s}$ 中断一次,在中断服务程序中重装定时初值,并将从SD卡中读取的WAV文件音频数据装入TLC5615进行D/A转换,即可实现按4kHz频率还原音频信号。由于音频数据为8位,因此TLC5615是10位DAC,为使TLC5615能输出 $0\sim 2.5\text{V}$ 的音频通道电压,需将从SD卡中读取的8位数据扩大2倍后再送给TLC5615进行D/A转换。为了使音频信号还原的过程平顺流畅,采用了双数据缓冲区,开辟两个512个元素的数组wav0和wav1,一个用于D/A转换,另一个用于装入SD卡读取的数据,二者轮流进行。

需要注意的是,本例不能在Proteus中实时仿真,需要利用Proteus的图文仿真功能将SD卡中的数据合成为WAV文件。在wavplayer.DSN文件中,打开如图9.12所示的AUDIO ANALYSIS图文仿真窗口,单击右键,再单击右键菜单中的“Simulate Graph”选项,启动图文仿真,即可生成如图9.13所示图文仿真结果,此时再单击AUDIO ANALYSIS图文仿真窗口右键菜单中的“Play Audio”选项,即可从PC机音箱中听到声音,同时还可以利用右键菜单中的“Edit Graph”选项,将图文仿真结果保存为WAV文件。

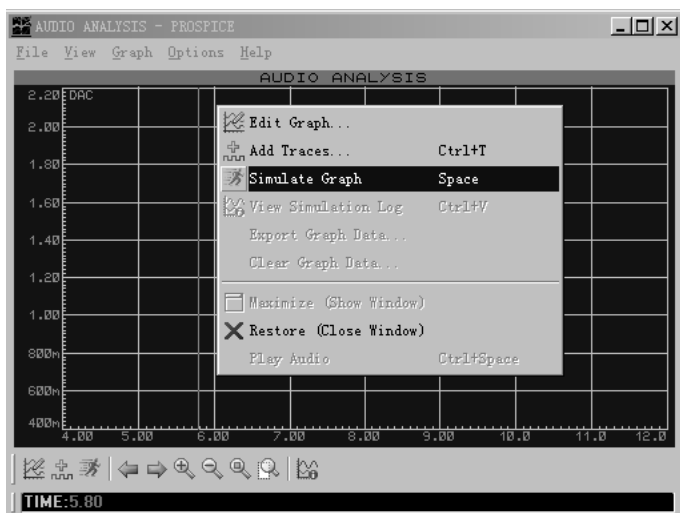


图9.12 AUDIO ANALYSIS图文仿真窗口

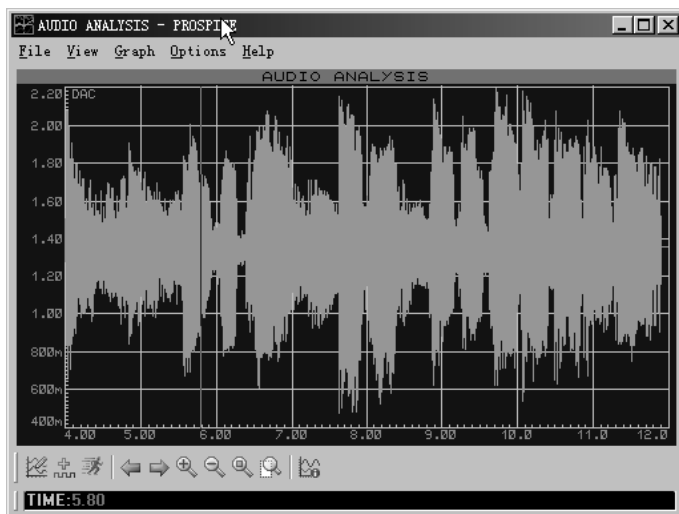


图9.13 AUDIO ANALYSIS图文仿真结果

例9-6 SD卡WAV音频播放器主程序main.c文件清单。

```
#include "reg52.h"
#include "intrins.h"
#include "stdio.h"
#include "uart.h"
#include "znfat/znFAT.h"
#include "myfun.h"

struct znFAT_Init_Args idata Init_Args; //初始化参数集合
struct FileInfo idata fileinfo;         //文件信息集合

static unsigned long WAV_LEN=0;         //WAV数据总长度
static unsigned int cnt=0;
static unsigned int len=0;               //每次读取WAV数据长度
static unsigned long N=0;

unsigned char bdata T0_Flag;             //定义定时器T0工作标志位
sbit RR0=T0_Flag^0;                     //WAV0与WAV1交替读取标志
sbit RR1=T0_Flag^1;
sbit FF0=T0_Flag^2;                     //WAV0与WAV1缓冲数据交替使用标志

unsigned char xdata wav0[512];           //WAV数据缓冲区
unsigned char xdata wav1[512];

sbit DIN = P1^7;                         //定义TLC5615引脚
sbit CLK = P1^5;
sbit CS = P1^6;

unsigned char bdata datH;                //定义TLC5615数据dat
unsigned char bdata datL;
sbit datH1=datH^1;                       //取出dat的各个位
sbit datH0=datH^0;
```

```

sbit datL7=datL^7;
sbit datL6=datL^6;
sbit datL5=datL^5;
sbit datL4=datL^4;
sbit datL3=datL^3;
sbit datL2=datL^2;
sbit datL1=datL^1;
sbit datL0=datL^0;

sbit LED=P1^4;                                     //LED指示灯

/***** TLC5615 D/A转换函数 *****/
void DaConv(unsigned int val){
    CLK=0;
    CS=1;
    _nop_();
    CS=0;
    datH=val>>8;  datL=val;
    DIN=datH1;
    CLK=1;
    CLK=0;
    DIN=datH0; CLK=0; CLK=1;
    DIN=datL7; CLK=0; CLK=1;
    DIN=datL6; CLK=0; CLK=1;
    DIN=datL5; CLK=0; CLK=1;
    DIN=datL4; CLK=0; CLK=1;
    DIN=datL3; CLK=0; CLK=1;
    DIN=datL2; CLK=0; CLK=1;
    DIN=datL1; CLK=0; CLK=1;
    DIN=datL0; CLK=0; CLK=1;
    DIN=0;      CLK=0; CLK=1; CLK=0; CLK=1;
    CLK=0;
    CS=1;
}

/***** 定时器T0初始化函数 *****/
void Timer0_Init(){
    TMOD|=0x01;
    TH0=0xfe; //250μs
    TL0=0x33; //T=1/F = 1/4000 = 250μs @22.11MHz
    ET0=1;
    EA=1;
    TR0=1;
}

/***** T0中断函数 *****/
void Timer0() interrupt 1{
    TR0=0;
    TH0=0xfe;
    TL0=0x33;
    if (FF0==0){
        //交替使用WAV0与WAV1缓冲数据

```



```

        DaConv(((unsigned int)wav0[cnt])<<1); //将wav数组中的数据扩大2倍后
        cnt++;
        if(cnt==len){
            RR0=1;
            FF0=1;
            cnt=0;
        }
    }
    if(FF0==1){ //交替使用WAV0与WAV1缓冲数据
        DaConv(((unsigned int)wav1[cnt])<<1);
        cnt++;
        if(cnt==len){
            RR1=1;
            FF0=0;
            cnt=0;
        }
    }
    TR0=1;
}

/***** int型数组转换为long型数据 *****/
long zc(){
    long s = 0;
    long s0 = wav0[40] & 0xff; // 最低位
    long s1 = wav0[41] & 0xff;
    long s2 = wav0[42] & 0xff;
    long s3 = wav0[43] & 0xff;
    s1 <= 8;
    s2 <= 16;
    s3 <= 24;
    s = s0 | s1 | s2 | s3 ;
    return s;
}

/*****LED闪动函数 *****/
void LED_Flash(){
    LED=~LED;
    delay(30);
}

/***** 主函数 *****/
void main(){
    unsigned int res=0;
    UART_Init();
    znFAT_Device_Init(); //存储设备初始化
    znFAT_Select_Device(0,&Init_Args); //选择设备
    while(!znFAT_Init()){
        printf("Suc. to init FS\r\n"); //文件系统初始化成功
        res=znFAT_Open_File(&fileinfo,"/yx.wav",0,1); //打开WAV文件
        if(!res){ //打开文件成功
            printf("Ready to play wav file: %s \r\n",fileinfo.File_Name);
            len=znFAT_ReadData(&fileinfo,0,50,wav0);

```

```

        WAV_LEN = zc(); //获取wav数据长度
        len=znFAT_ReadData(&fileinfo,44,512,wav0); //装入wav0数据
        N=N+512;
        len=znFAT_ReadData(&fileinfo,N,512,wav1); //装入wav1数据
        N=N+512;
        WAV_LEN = WAV_LEN-1024;
    }
    else{
        printf("Fail to open file, Err Code: %u",res); //打开文件失败
        break;
    }
    Timer0_Init(); //定时器初始化
    while(1){
        if(RR1==1){
            if(WAV_LEN>=512){
                len=znFAT_ReadData(&fileinfo,N,512,wav1);
            }
            else{
                len=znFAT_ReadData(&fileinfo,N,WAV_LEN,wav1);
                //到最后一帧数据
                printf("That's the end of wav file \r\n");
                //WAV文件结束
                TR0=0; N=0; LED=0;break; //停止
            }
            RR1=0;N=N+512;WAV_LEN=WAV_LEN-len;
        }
        if(RR0==1){
            if(WAV_LEN>=512){
                len=znFAT_ReadData(&fileinfo,N,512,wav0);
            }
            else{
                len=znFAT_ReadData(&fileinfo,N,WAV_LEN,wav0);
                //到最后一帧数据
                printf("That's the end of wav file \r\n");
                //WAV文件结束
                TR0=0; N=0; LED=0;break; //停止
            }
            RR0=0;N=N+512;WAV_LEN=WAV_LEN-len;
        }
        znFAT_Flush_FS(); //刷新文件系统
    }
    znFAT_Close_File(&fileinfo); //关闭文件
    while(1); //停止播放
}
while(1){
    LED_Flash(); //文件读取出错
}
}

```

8051指令表

表中所用到的符号和含义如下:

P	程序状态字寄存器PSW中的奇偶标志位
OV	程序状态字寄存器PSW中的溢出标志
AC	程序状态字寄存器PSW中的辅助进位标志
CY	程序状态字寄存器PSW中的进位标志
addr11	11位地址
addr16	16位地址
a ₁₀ a ₉ a ₈	11位地址中的最高3位
bit	位地址
rel	相对偏移量, 为8位有符号数(补码形式)
direct	直接地址指出的单元内容
#data	立即数
Rn	工作寄存器Rn ($n=0\sim7$)
(Rn)	工作寄存器Rn的内容
A	累加器
(A)	累加器内容
Ri	$i=0$ 或 1
(Ri)	R0或R1的内容
((Ri))	R0或R1指出单元的内容
X	某一个寄存器
(X)	某一个寄存器内容
((X))	某一个寄存器指出的单元内容
→	数据传送方向
⊕	逻辑异或
∧	逻辑与
∨	逻辑或
✓	对标志产生影响
×	不影响标志

表1 算术运算指令

算术运算指令								
十六进制代码	助 记 符	功 能	对标志影响				字节数	周期数
			P	OV	AC	CY		
28~2F	ADD A, Rn	$(A)+(Rn) \rightarrow A$	√	√	√	√	1	1
25	ADD A, direct	$(A)+(direct) \rightarrow A$	√	√	√	√	2	1
26~27	ADD A, @Ri	$(A)+((Ri)) \rightarrow A$	√	√	√	√	1	1
24	ADD A, #data	$(A)+data \rightarrow A$	√	√	√	√	2	1
38~3F	ADDC A, Rn	$(A)+(Rn)+CY \rightarrow A$	√	√	√	√	1	1
35	ADDC A, direct	$(A)+(direct)+CY \rightarrow A$	√	√	√	√	2	1
36~37	ADDC A, @Ri	$(A)+((Ri))+CY \rightarrow A$	√	√	√	√	1	1
34	ADDC A, #data	$(A)+data+CY \rightarrow A$	√	√	√	√	2	1
98~9F	SUBB A, Rn	$(A)-(Rn)-CY \rightarrow A$	√	√	√	√	1	1
95	SUBB A, direct	$(A)-(direct)-CY \rightarrow A$	√	√	√	√	2	1
96~97	SUBB A, @Ri	$(A)-((Ri))-CY \rightarrow A$	√	√	√	√	1	1
94	SUBB A, #data	$(A)-data-CY \rightarrow A$	√	√	√	√	2	1
04	INC A	$(A)+1 \rightarrow A$	√	×	×	×	1	1
08~0F	INC Rn	$(Rn)+1 \rightarrow (Rn)$	×	×	×	×	1	1
05	INC direct	$(direct)+1 \rightarrow direct$	×	×	×	×	2	1
06~07	INC @Ri	$((Ri))+1 \rightarrow (Ri)$	×	×	×	×	1	1
A3	INC DPTR	$(DPTR)+1 \rightarrow DPTR$	×	×	×	×	1	2
14	DEC A	$(A)-1 \rightarrow A$	√	×	×	×	1	1
18~1F	DEC Rn	$(Rn)-1 \rightarrow Rn$	×	×	×	×	1	1
15	DEC direct	$(direct)-1 \rightarrow direct$	×	×	×	×	2	1
16~17	DEC @Ri	$((Ri))-1 \rightarrow (Ri)$	×	×	×	×	1	1
A4	MUL AB	$(A)*(B) \rightarrow BA$	√	√	×	√	1	4
84	DIV AB	$(A)/(B) \rightarrow AB$	√	√	×	√	1	4
D4	DA A	对(A)进行BCD码调整	√	√	√	√	1	1

表2 逻辑运算指令

逻辑运算指令								
十六进制代码	助 记 符	功 能	对标志影响				字节数	周期数
			P	OV	AC	CY		
58~5F	ANL A,Rn	$(A)\wedge(Rn) \rightarrow A$	√	×	×	×	1	1
55	ANL A,direct	$(A)\wedge(direct) \rightarrow A$	√	×	×	×	2	1
56~57	ANL A,@Ri	$(A)\wedge((Ri)) \rightarrow A$	√	×	×	×	1	1
54	ANL A,#data	$(A)\wedge data \rightarrow A$	√	×	×	×	2	1
52	ANL direct,A	$(direct)\wedge(A) \rightarrow direct$	×	×	×	×	2	1
53	ANL direct,#data	$(direct)\wedge data \rightarrow direct$	×	×	×	×	3	2
48~4F	ORL A,Rn	$(A)\vee(Rn) \rightarrow A$	√	×	×	×	1	1
45	ORL A,direct	$(A)\vee(direct) \rightarrow A$	√	×	×	×	2	1
46~47	ORL A,@Ri	$(A)\vee((Ri)) \rightarrow A$	√	×	×	×	1	1
44	ORL A,#data	$(A)\vee data \rightarrow A$	√	×	×	×	2	1

(续表)

逻辑运算指令								
十六进制代码	助 记 符	功 能	对标志影响				字节数	周期数
			P	OV	AC	CY		
42	ORL direct,A	$(\text{direct}) \vee (A) \rightarrow \text{direct}$	×	×	×	×	2	1
43	ORL direct,#data	$(\text{direct}) \vee \text{data} \rightarrow \text{direct}$	×	×	×	×	3	2
68~6F	XRL A,Rn	$(A) \oplus (Rn) \rightarrow A$	√	×	×	×	1	1
65	XRL A,direct	$(A) \oplus (\text{direct}) \rightarrow A$	√	×	×	×	2	1
66~67	XRL A,@Ri	$(A) \oplus ((Ri)) \rightarrow A$	√	×	×	×	1	1
64	XRL A,#data	$(A) \oplus \text{data} \rightarrow A$	√	×	×	×	2	1
62	XRL direct,A	$(\text{direct}) \oplus (A) \rightarrow \text{direct}$	×	×	×	×	2	1
63	XRL direct,#data	$(\text{direct}) \oplus \text{data} \rightarrow \text{direct}$	×	×	×	×	3	2
E4	CLR A	$0 \rightarrow A$	√	×	×	×	1	1
F4	CPL A	$\overline{(A)} \rightarrow A$	×	×	×	×	1	1
23	RL A	A循环左移一位	×	×	×	×	1	1
33	RLC A	A带进位循环左移一位	√	×	×	√	1	1
03	RR A	A循环右移一位	×	×	×	×	1	1
13	RRC A	A带进位循环右移一位	√	×	×	√	1	1
C4	SWAP A	A半字节交换	×	×	×	×	1	1

表3 数据传送指令

数据传送指令								
十六进制代码	助 记 符	功 能	对标志影响				字节数	周期数
			P	OV	AC	CY		
E8~EF	MOV A,Rn	$(Rn) \rightarrow A$	√	×	×	×	1	1
E5	MOV A,direct	$(\text{direct}) \rightarrow A$	√	×	×	×	2	1
E6~E7	MOV A,@Ri	$((Ri)) \rightarrow A$	√	×	×	×	1	1
74	MOV A,#data	$\text{data} \rightarrow A$	√	×	×	×	2	1
F8~FF	MOV Rn,A	$(A) \rightarrow Rn$	×	×	×	×	1	1
A8~AF	MOV Rn,direct	$(\text{direct}) \rightarrow Rn$	×	×	×	×	2	2
78~7F	MOV Rn,#data	$\text{data} \rightarrow Rn$	×	×	×	×	2	1
F5	MOV direct,A	$(A) \rightarrow \text{direct}$	×	×	×	×	2	1
88~8F	MOV direct,Rn	$(Rn) \rightarrow \text{direct}$	×	×	×	×	2	2
85	MOV direct1,direct2	$(\text{direct2}) \rightarrow \text{direct1}$	×	×	×	×	3	2
86~87	MOV direct,@Ri	$((Ri)) \rightarrow \text{direct}$	×	×	×	×	2	2
75	MOV direct,#data	$\text{data} \rightarrow \text{direct}$	×	×	×	×	3	2
F6~F7	MOV @Ri,A	$(A) \rightarrow (Ri)$	×	×	×	×	1	1
A6~A7	MOV @Ri,direct	$(\text{direct}) \rightarrow (Ri)$	×	×	×	×	2	2
76~77	MOV @Ri,#data	$\text{data} \rightarrow (Ri)$	×	×	×	×	2	1
90	MOV DPTR,#data16	$\text{data16} \rightarrow \text{DPTR}$	×	×	×	×	3	2
93	MOVC A,@A+DPTR	$((A)+(\text{DPTR})) \rightarrow A$	√	×	×	×	1	2
83	MOVC A,@A+PC	$((A)+(\text{PC})) \rightarrow A$	√	×	×	×	1	2

(续表)

数据传送指令								
十六进制代码	助记符	功能	对标志影响				字节数	周期数
			P	OV	AC	CY		
E2~E3	MOVX A,@Ri	((Ri)+(P2))→A	√	×	×	×	1	2
E0	MOVX A,@DPTR	((DPTR))→A	√	×	×	×	1	2
F2~F3	MOVX @Ri,A	(A)→(Ri)+(P2)	×	×	×	×	1	2
F0	MOVX @DPTR,A	(A)→(DPTR)	×	×	×	×	1	2
C0	PUSH direct	(SP)+1→SP (direct)→(SP)	×	×	×	×	2	2
D0	POP direct	((SP))→direct (SP)-1→SP	×	×	×	×	2	2
C8~CF	XCH A,Rn	(A) ↔ (Rn)	√	×	×	×	1	1
C5	XCH A,direct	(A) ↔ (direct)	√	×	×	×	2	1
C6~C7	XCH A,@Ri	(A) ↔ ((Ri))	√	×	×	×	1	1
D6~C7	XCHD A,@Ri	(A) ₀₋₃ ↔ ((Ri)) ₀₋₃	√	×	×	×	1	1

表4 控制转移指令

控制转移指令								
十六进制代码	助记符	功能	对标志影响				字节数	周期数
			P	OV	AC	CY		
a ₁₀ a ₉ a ₈ 1 0001 (二进制代码)	ACALL,addr11	(PC)+2→PC, (SP)+1→SP, (PCL)→(SP), (SP)+1→SP, (PCH)→(SP), addr11→PC	×	×	×	×	2	2
12	LCALL addr16	(PC)+3→PC, (SP)+1→SP, (PCL)→(SP), (SP)+1→SP, (PCH)→(SP), addr16→PC	×	×	×	×	3	2
22	RET	((SP))→PCH, (SP)-1→SP, ((SP))→PCL, (SP)-1→SP	×	×	×	×	1	2
32	RETI	((SP))→PCH, (SP)-1→SP, ((SP))→PCL, (SP)-1→SP 从中断返回	×	×	×	×	1	2
a ₁₀ a ₉ a ₈ 0 0001 (二进制代码)	AJMP addr11	(PC)+2→PC, addr11→PC	×	×	×	×	2	2
02	LJMP addr16	addr16→PC	×	×	×	×	3	2
80	SJMP rel	(PC)+2→PC, (PC)+rel→PC	×	×	×	×	2	2
73	JMP @A+DPTR	(A)+(DPTR)→PC	×	×	×	×	1	2
60	JZ rel	(PC)+2→PC, 若(A)=0, (PC)+rel→PC	×	×	×	×	2	2
70	JNZ rel	(PC)+2→PC, 若(A)≠0, 则(PC)+rel→PC	×	×	×	×	2	2
40	JC rel	(PC)+2→PC, 若CY=1, 则(PC)+rel→PC	×	×	×	×	2	2
50	JNC rel	(PC)+2→PC, 若CY=0, 则(PC)+rel→PC	×	×	×	×	2	2

(续表)

控制转移指令								
十六进制代码	助 记 符	功 能	对标志影响				字节数	周期数
			P	OV	AC	CY		
20	JB bit,rel	(PC)+3→PC, 若(bit)=1, 则(PC)+rel→PC	×	×	×	×	3	2
30	JNB bit,rel	(PC)+3→PC, 若(bit)=0, 则(PC)+rel→PC	×	×	×	×	3	2
10	JBC bit,rel	(PC)+3→PC, 若(bit)=1, 则0→bit, (PC)+rel→PC	×	×	×	×	3	2
B5	CJNE A,direct,rel	(PC)+3→PC, 若(A)不等于(direct), 则(PC)+rel→PC, 若(A)<(direct), 则1→CY	×	×	×	√	3	2
B4	CJNE A,#data,rel	(PC)+3→PC, 若(A)不等于data, 则(PC)+rel→PC, 若(A)<data, 则1→CY	×	×	×	√	3	2
B8~BF	CJNE Rn,#data,rel	(PC)+3→PC, 若(Rn)不等于data, 则(PC)+rel→PC, 若(Rn)<data, 则1→CY	×	×	×	√	3	2
B6~B7	CJNE @Ri,#data,rel	(PC)+3→PC, 若(Ri)不等于data, 则(PC)+rel→PC, 若(Ri)<data, 则1→CY	×	×	×	√	3	2
D8~DF	DJNZ Rn,rel	(PC)+2→PC, (Rn)-1→Rn 若(Rn)不等于0, 则(PC)+rel→PC	×	×	×	×	2	2
D5	DJNZ direct,rel	(PC)+3→PC, (direct)-1→direct, 若(direct)不等于0, 则(PC)+rel→PC	×	×	×	×	3	2
00	NOP	空操作	×	×	×	×	1	1

表5 位操作指令

位 操 作 指 令								
十六进制代码	助 记 符	功 能	对标志影响				字节数	系统时钟数
			P	OV	AC	CY		
C3	CLR C	0→CY	×	×	×	√	1	1
C2	CLR bit	0→bit	×	×	×	×	2	1
D3	SETB C	1→CY	×	×	×	√	1	1
D2	SETB bit	1→bit	×	×	×	×	2	1
B3	CPL C	$\overline{CY} \rightarrow CY$	×	×	×	√	1	1
B2	CPL bit	$\overline{bit} \rightarrow bit$	×	×	×	×	2	1
82	ANL C,bit	$(CY) \wedge (bit) \rightarrow CY$	×	×	×	√	2	2
B0	ANL C _i ,bit	$(CY) \wedge \overline{(bit)} \rightarrow CY$	×	×	×	√	2	2
72	ORL C,bit	$(CY) \vee (bit) \rightarrow CY$	×	×	×	√	2	2
A0	ORL C _i ,bit	$(CY) \vee \overline{(bit)} \rightarrow CY$	×	×	×	√	2	2
A2	MOV C,bit	(bit)→CY	×	×	×	√	2	1
92	MOV bit,C	CY→bit	×	×	×	×	2	2

Proteus中的常用元器件

元 器 件 名	中 文 注 释	元 器 件 名	中 文 注 释
80C51	8051单片机	RELAY	继电器
AT89C52	Atmel 8052单片机	ALTERNATOR	交互式交流电压源
CRYSTAL	晶体振荡器	POT-LIN	交互式电位计
CERAMIC22P	陶瓷电容	CAP-VAR	可变电容
CAP	电容	CELL	单电池
CAP-ELEC	通用电解电容	BATTERY	电池组
RES	电阻	AREIAL	天线
RX8	8电阻排	PIN	单脚终端接插针
RESPACK-8	带公共端的8电阻排	LAMP	动态灯泡模型
MINRES5K6	5k6电阻	TRAFFIC	动态交通灯模型
74LS00	四2输入与非门	SOUNDER	压电发声模型
74LS164	8位并出串行移位寄存器	SPEAKER	喇叭模型
74LS244	8同相三态输出缓冲器	7805	5V,1A稳压器
74LS245	8同相三态输出收发器	78L05	5V,100mA稳压器
NOR	二输入或非门	LED-GREEN	绿色发光二极管
OR	二输入或门	LED-RED	红色发光二极管
XOR	二输入异或门	LED-YELLOW	黄色发光二极管
NAND	二输入与非门	MAX7219	串行8位LED显示驱动器
AND	二输入与门	7SEG-BCD	七段BCD数码管
NOT	数字反相器	7SEG-DIGITAL	七段数码管
COMS	COMS系列	7SEG-COM-CAT-GRN	七段共阴极绿色数码管
4001	双2输入或非门	7SEG-COM-AN-GRN	七段共阳极绿色数码管
4052	双4通道模拟开关	7SEG-MPX6-CA	6位七段共阳极红色数码管
4511	BCD-7段锁存/解码/驱动器	7SEG-MPX6-CC	6位七段共阴极红色数码管
DIODE-TUN	通用沟道二极管	MATRIX-5×7-RED	5×7点阵红色LED显示器

(续表)

元 器 件 名	中 文 注 释	元 器 件 名	中 文 注 释
UF4001	二极管急速整流器	MATRIX-8×8-BLUE	8×7点阵蓝色LED显示器
1N4148	小信号开关二极管	AMPIRE128×64	128×64图形LCD
SCR	通用晶闸管整流器	LM016L	16×2字符LCD
TRIAC	通用三端双向晶闸管开关	555	定时器/振荡器
MOTOR	简单直流电动机	NPN	通用NPN型双极性晶体管
MOTOR-STEPPER	动态单极性步进电动机	PNP	通用PNP型双极性晶体管
MOTOR-SERVO	伺服电动机	PMOSFET	通用P型金属氧化物场效应管
COMPIN	COM口物理接口模型	2764	8k×8 EPROM存储器
CONN-D9M	9针D型连接器	6264	8k×8静态RAM存储器
CONN-D9F	9孔D型连接器	24C04	4K位I ² C EEPROM存储器
BUTTON	按钮	ADC0808	8位8通道A/D转换器
SWITCH	带锁存开关	DAC0832	8位D/A转换器
SW-SPST-MOM	非锁存开关	DS1302	日历时钟

参考文献

- [1] C51 Compiler User's Guide. Keil Elektronik GmbH. and Keil Software, Inc. 2010.
- [2] Getting Started and Creating Applications User's Guide. Keil Elektronik GmbH., and Keil Software, Inc. 2010.
- [3] 徐爱钧. 单片机高级语言C51应用程序设计. 电子工业出版社, 2000.
- [4] 徐爱钧, 彭秀华. Keil Cx51 V7.0单片机高级语言编程与 μ Vision2应用实践(第2版). 电子工业出版社, 2008.
- [5] 徐爱钧. 单片机原理与应用——基于Proteus虚拟仿真技术. 机械工业出版社, 2011.
- [6] 徐爱钧, 徐阳. 智能化测量控制仪表原理与设计(第3版). 北京航空航天大学出版社, 2012.
- [7] 徐爱钧, 徐阳. Keil C51单片机高级语言应用编程与实践. 电子工业出版社, 2013.
- [8] 徐爱钧. STC15增强型8051单片机C语言编程与应用. 电子工业出版社, 2014.
- [9] 徐爱钧. 单片机原理实用教程——基于Proteus虚拟仿真(第3版). 电子工业出版社, 2014.

